POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

5G Edge Offloading for Drone Swarms

Supervisors

Prof. Fulvio RISSO

Ing. Jacopo MARINO

MSc. Daniele CACCIABUE

Candidate

Sasha ALGISI

July 2025

Summary

Over the past few years, we have experienced a significant evolution in cloud technologies and paradigms, such as the "Edge Computing" model, which allows to deliver quick responses by offering services at the network's edge, thus significantly reducing the latency in favor of responsiveness. In order to find a solution that could efficiently leverage the computational capability present at the network's edge, the “Flexible, scaLable, secUre, and decentralIseD Operating System” (FLUIDOS) European project has been proposed, which has as its main goal the creation of a seamless computing continuum that can be used to integrate heterogeneous devices, thus creating a single virtual cluster between the IoT devices, such as drones, and the network's edge. This work aims at enhancing FLUIDOS capabilities by providing the appropriate support for ensuring seamless operations even in fluctuating network conditions, like in 5G networks, through the “Distributed Edge Analytics Service” (DEAS) open call. The thesis will discuss the design and implementation of the solution required to achieve an autonomous, adaptive, and efficient workload distribution that allows to dynamically offload tasks between drones and the edge by using the quality of the 5G channel as a decision criterion, thus ensuring a responsive computing continuum. As a conclusion, the thesis will evaluate the performance of the proposed solution and will discuss the possible future directions of the project.

Acknowledgements

“Ai miei genitori, per avermi sempre sostenuto nella vita davanti alle difficoltà.”

Table of Contents

List of Figures

Acronyms

FLUIDOS	Flexible, scaLable, secUre, and decentralIseD Operating System
REAR	REsource Advertisement and Reservation
DEAS	Distributed Edge Analytics Service
K8s	Kubernetes
ROS	Robot Operating System
VM	Virtual Machine
MEC	Mobile Edge Cloud
CNCF	Cloud Native Computing Foundation
UE	User Equipment
CNR	Consiglio Nazionale delle Ricerche

Chapter 1

Introduction

Nowadays, Cloud Computing has become one of the most important technologies in our society due to its great capability of delivering multiple services like storage, infrastructures and software products to companies and final users through the internet by offering, at the same time, benefits in terms of scalability, flexibility, reduced costs, security and many others. This paradigm has some limitations though, like for example the fact that the data centers that are typically employed for cloud services are centralized entities, distant from the users; this can lead to several problems as increased latency and lower bandwidth, especially for IoT devices, which typically execute real-time tasks that require great responsiveness. For this reason, the **Edge Computing** model has been introduced, which is a paradigm that has as its main goal the delivering of quick responses by offering services at the network's edge through the addition of multiple smaller data centers located near the users instead of just relying on a bigger (and distant) data center, this allows to significantly reduce the latency, as now services are also offered near the users, by increasing the available bandwidth and the overall responsiveness. There is one problem though, which is the fact that now the architecture is far more complex than it was before, as we have multiple layers to consider now (Figure 1.1), furthermore, it is worth mentioning that these layers are formed by devices with different hardware running different operating systems and with different purposes, this complicates the scenario even more; for this reason, the **Computing continuum** paradigm has been conceived.

1.1 The importance of a responsive computing continuum

The Computing continuum is the paradigm that integrates, organizes and considers the computing and network resources across different infrastructures (endpoint devices, edge nodes, cloud data centers, and the networks connecting them) for deploying workloads, instead of relying on a specific infrastructure [1]. As already mentioned, this paradigm becomes particularly relevant when considering the fact that the infrastructure's layers are composed by heterogeneous devices, hence, in traditional scenarios, resources' allocation and management can be challenging. The Computing continuum allows to consider all these different layers as a single entity, where computational tasks can be deployed without paying particular attention to the diversities of the devices or where the tasks are executed, this can eventually lead to a better utilization of the resources inside the system by creating a dynamic and fluid environment. The biggest challenge in the realization of what has been previously discussed is the dynamicity of the network, as the quality of the channels can change over time in an unpredictable way, an example of this are wireless networks, which are more prone to this eventuality; this problematic further emphasizes the importance of making the existing Computing continuum responsive to the network conditions, so that computational tasks can be optimally allocated between different layers by considering as an additional decision criterion the eventual changes in the channels' quality, thus making sure that the workload is dynamically offloaded from one layer to the other without any service disruption.

1.2 Goal of the thesis

The goal of this thesis is to create an infrastructure capable of ensuring seamless operations in scenarios where tasks are periodically offloaded between different layers, even during fluctuating network conditions. This work will mainly focus on developing a solution that can dynamically offload the workload of IoT devices (such as drones) between them and the network's edge in 5G networks, depending on the network's quality; this will make sure that the system's computational resources usage is always kept optimal, as this allows to lighten the drones' workload when the conditions are met, by creating a responsive computing continuum. The first step is to conceive a valid strategy that allows the system to understand whether its the case to execute the workload on the drones or on the network's edge by discussing the advantages and disadvantages of said strategy, the next step is to design the required architecture needed to reach this goal by also analyzing various solutions for the implementation of the offloading process. The last step is to evaluate the obtained performance by comparing it with possible alternative strategies; the validity of the solution will take into account characteristics like efficiency, adaptability, absence of manual interventions and time cost.

1.3 Thesis structure

1. **Chapter 2:** This chapter introduces the edge computing paradigm by discussing its current state of the art.
2. **Chapter 3:** This chapter explains what Kubernetes is and its main concepts.
3. **Chapter 4:** This chapter introduces Liqo and how it can help the creation of dynamic and seamless Kubernetes multi-cluster topologies.
4. **Chapter 5:** This chapter provides an introduction to the FLUIDOS project and its goals.
5. **Chapter 6:** This chapter shows the implementation of an autonomous and adaptive solution that allows to dynamically offload the workload from one cluster to the other, even in fluctuating network conditions.
6. **Chapter 7:** This chapter shows how the system behaves in a real scenario and its performances.
7. **Chapter 8:** This chapter concludes the thesis by discussing the obtained results and its possible future directions.

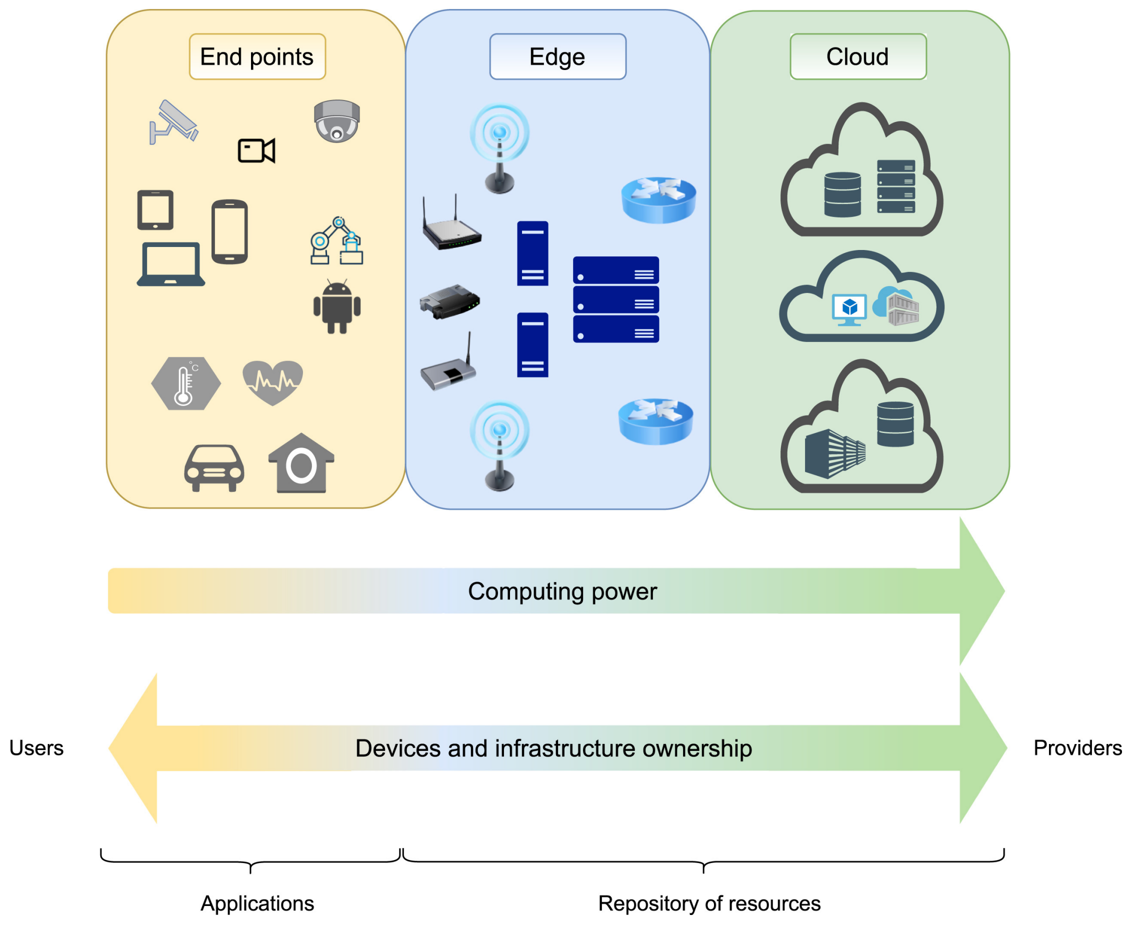


Figure 1.1: Different layers in a computing continuum scenario [1]

Chapter 2

Edge Computing

This chapter will present a more in-depth overview about the edge computing paradigm by explaining why it matters and its main benefits. As a conclusion, it will analyze the paradigm's current state of the art by discussing some of the most relevant works regarding this subject.

2.1 The need of an edge computing model

As already explained in Chapter 1, the edge computing paradigm represents an extension to the cloud computing technology that allows to deliver services at the network's edge, hence near the final users; this is mainly done by deploying additional smaller data centers near them, so that requests may be served quicker (Figure 2.1). This approach is particularly relevant for real-time applications, which are the ones that are typically executed on IoT devices, as the smaller latency allows to achieve an increased responsiveness for said applications. There are several benefits that the introduction of this paradigm offers, let us discuss some of the most important ones:

- **Reduced latency:** the services are now located at the network's edge, thus quicker responses are guaranteed by making sure to place the data processing closer to where this data is generated.
- **Increased Bandwidth:** considering that now user devices can make requests to the nearest data center instead of having all of them visiting the remote one, the available bandwidth will be greater due to fact that less devices will make requests to the same servers.
- **Increased security:** by keeping users' sensitive information closer to devices instead of transmitting them to the main data center by passing through the network multiple times, an improved level of security can be achieved.

- **Increased scalability:** due to the fact that now there are multiple smaller data centers available to serve requests from users, now more devices can be served without increasing the overall congestion.

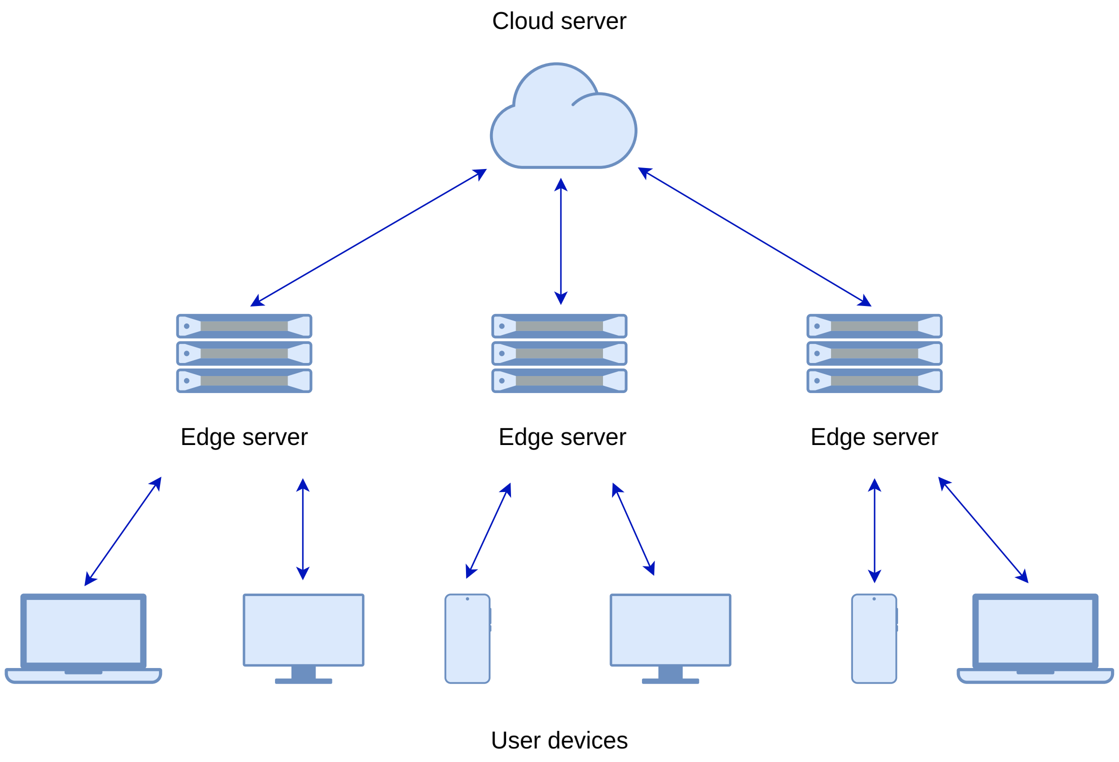


Figure 2.1: Typical edge computing architecture

Of course, the introduction of this idea will inevitably create additional challenges, as now tasks are geographically distributed between different locations, thus the overall management is more complex; furthermore it should be considered that its not simple to effectively understand where these tasks should be placed in order to obtain the best performance possible, in fact, for example, some of these smaller data centers' computational resources could be already used for other tasks, therefore adding even more workload would overload them. Another important aspect to consider is that the network itself is unpredictable, thus communications between different locations could be difficult, especially when we consider fluctuating network conditions (as for example in 5G networks). The next section will present some relevant works in the field that tried to propose different solutions and strategies for the creation of an efficient edge computing infrastructure that could reach the required level of performance by avoiding, at the same time, some of the aforementioned problems.

2.2 State of the art

When we consider edge computing, it is important to note that real-time applications are the ones that can benefit the most from this paradigm, as reduced latencies can be achieved most of the times for edge devices running said applications. There is a problem though, which is the fact that the implementation of a system that can satisfy the latency requirements of these applications can be very challenging, an example of this are robots, as they often execute time-constrained tasks that require an optimal level of performance.

2.2.1 FogROS

Some of the aforementioned problems have been addressed by Chen et al. [2] through the introduction of **FogROS**, a framework that acts as an extension to the "**Robot Operating System**" (ROS); this tool allows to deploy robot software components to the cloud with minimal effort, hence additional computing resources like CPU cores, GPUs, FPGAs, or TPUs can be obtained in order to improve the overall performance, this is particularly relevant for real-time tasks that require minimal latency. Ichnowski et al. [3] later presented **FogROS2** for ROS2, which increases even more performance, security and automation capabilities compared to FogROS by enabling cloud and fog computing integration for robots in a simple way; furthermore it uses VMs instead of K8s (which is more complex). Chen et al. [4] expanded FogROS2's capabilities by introducing **FogROS2-SGC**, an extension of the former that allows to connect robots that are geographically distant between them securely without any code modification; this was not possible before, as FogROS2 originally assumed that all the robots were locally connected.

2.2.2 Further optimizations

Other works, like Chebaane et al. [5], addressed the problem of offloading time-critical application tasks, as live migration is an important process that has to be kept efficient; specifically, this research proposed a solution that allows to offload these tasks from the application initiator to surrounding Fog nodes by using **Docker containers and Checkpointing**, thus allowing to restrict the offloading to only necessary steps in only two message rounds. Some researches tried to implement innovative strategies that could improve the performance by minimizing the typical problems that are often present during network transmissions, like Anand et al. [6], where, in order to overcome some limitations of serverless computing, such as communication and bandwidth issues, an algorithm that uses different **work-sharing techniques** has been presented, one that allows to achieve cost and execution time optimizations. One of the biggest challenges for latency-sensitive applications is user mobility, as this may cause several service interruptions for said applications; this problem has been addressed by Doan et al. [7] by introducing a framework that aims at separating MEC applications into two different components, which are the processing and the state management ones. This can be done by using a **distributed key-value store** that can ensure seamless service continuity by enabling state synchronization across MEC servers, hence reducing downtime by half in most of the cases. Similarly, Machen et al. [8] addressed the challenge of minimizing service downtime and migration time for MEC applications by developing a **layered framework** that can break down these applications into multiple layers, the main advantage is that only missing layers have to be transferred to other locations; furthermore, this framework is applicable to both VMs and containers. This thesis differentiates itself from other researches by developing a solution that mainly focuses on using the network's quality in order to understand the best placement for running tasks (Section 1.2), as this allows to optimize the overall energy consumption of IoT devices by offloading their workload to the edge when the appropriate network conditions are met.

Chapter 3

Kubernetes

This chapter will present Kubernetes by discussing its origins, architecture and some of its most relevant scheduling concepts, which are the foundations that allow to understand the implementation process of this work. In conclusion, Prometheus will be introduced by explaining its purpose and advantages.

3.1 Kubernetes history

Originally, Google created the **Borg** system around 2003-2004, which was a large-scale internal cluster management system that "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines" [9]. On June 6, 2014 Google presented **Kubernetes** as on open-source version of Borg; it was created by Joe Beda, Brendan Burns, and Craig McLuckie, and other engineers at Google. As one could imagine, its development and design were heavily influenced by Borg, additionally many of its top contributors had previously worked on Borg itself. One of the most notable differences between Kubernetes and its predecessor is the fact that while the latter was written in C++, the former has been written by using the Go language. On July 21, 2015 Kubernetes 1.0 was released, furthermore Google formed alongside the Linux Foundation the "**Cloud Native Computing Foundation**" (CNCF) [10], this eventually led to the adoption of Kubernetes by nearly every big company, thus becoming the de-facto standard for container orchestration [11].

3.2 Kubernetes architecture

A Kubernetes cluster (Figure 3.1) consists of a control plane plus a set of worker machines, called **nodes**, that run containerized applications. Every cluster needs at least one worker node in order to run **Pods**. The worker node(s) host the Pods that are the components of the application workload. The **control plane** manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

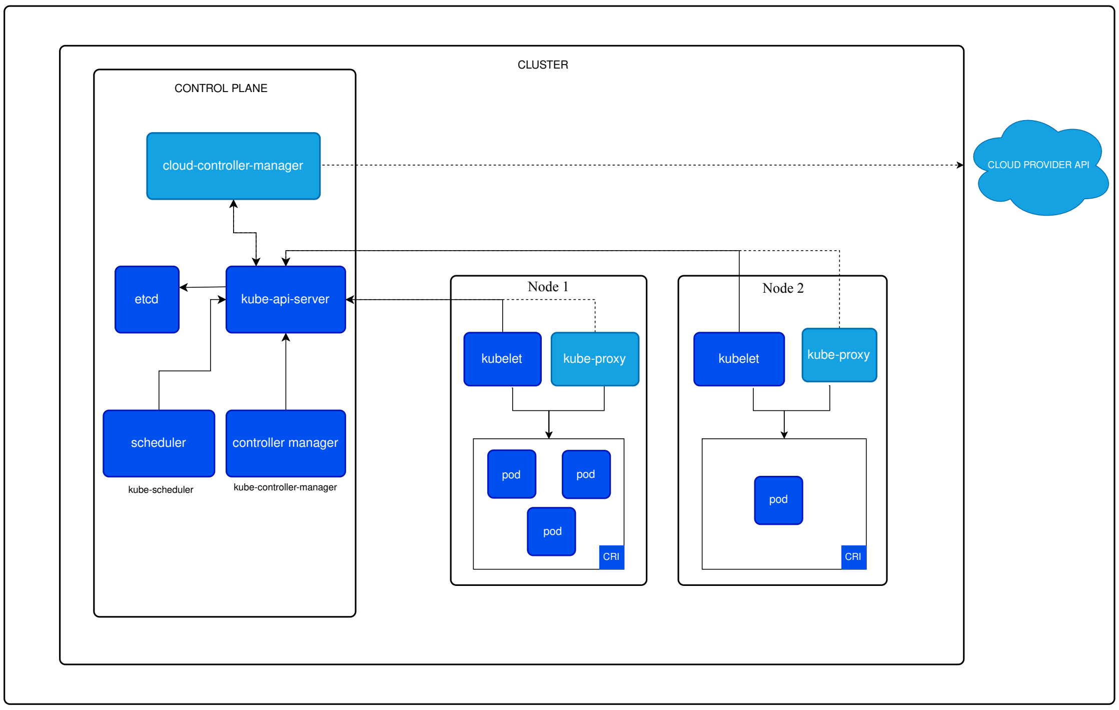


Figure 3.1: Kubernetes architecture [12]

3.2.1 Control plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (like starting up a new Pod). Even though control plane components can be run on any machine in the cluster, for the sake of simplicity, setup scripts typically start all of these components on the same machine, which is not the same one where user containers run.

kube-api-server

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API, it also serves as the front end for the Kubernetes control plane itself, as it is the one that receives incoming REST requests. `kube-api-server` is designed to scale horizontally, this means that it scales by deploying more instances, furthermore, several instances of this component can be created in order to balance traffic between them.

etcd

`etcd` is a consistent, distributed and highly-available key-value store used as Kubernetes' backing store for all cluster data. It is based on the Raft consensus algorithm, where, if there are several nodes, a leader node is selected through a vote; this leader will hold all the key-value pairs. After this is done, whenever a new request is made to any node, it will be forwarded to the leader, which will handle said request, make the changes in the key-value pair and inform the follower nodes; these other nodes will make the corresponding changes to match the leader node. As long as the majority of the nodes in the system are functional, a new leader can always be elected, thus guaranteeing service continuity in case of breakdowns. `kube-api-server` is the only component that can interact with `etcd`.

kube-scheduler

`kube-scheduler` is the control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on. The main factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager

Component that runs controller processes. It is a daemon that acts as a continuous control loop by comparing the desired state of the cluster, derived from the objects specifications, with the current one, which is read from `etcd`. Even though, from a logical point of view, each controller is a separate process, they are all compiled into a single binary and run in a single process, this is done in order to reduce complexity. Some of these controllers are:

- **Node controller**: responsible for noticing and responding when nodes go down.
- **Job controller**: watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- **EndpointSlice controller**: populates EndpointSlice objects (to provide a link between Services and Pods).
- **ServiceAccount controller**: create default ServiceAccounts for new namespaces.

cloud-controller-manager

`cloud-controller-manager` runs controllers that are specific to the underlying cloud providers. In order to separate out the components that interact with other cloud platforms from components that only interact with the local cluster, it can link the former into the cloud provider's API; this will make sure that the cloud vendor's code and the Kubernetes code can evolve independently, as, originally, the Kubernetes code was depended on cloud-provider-specific code for its functionalities. Similarly to `kube-controller-manager`, this component combines several logically independent control loops into a single binary that you run as a single process. Some controllers that can have cloud provider dependencies are:

- **Node controller**: checks the cloud provider to determine if a node has been deleted in the cloud after it stops responding.
- **Route controller**: responsible of setting up network routes in the underlying cloud infrastructure.
- **Service controller**: for creating, updating and deleting cloud provider load balancers.

3.2.2 Node components

Node components run on every node, maintaining running Pods and providing the Kubernetes runtime environment.

kubelet

An agent that runs on each node in the cluster, its purpose is to make sure that containers are running in a Pod. `kubelet` takes a set of `PodSpecs` from the API server and ensures that the containers described in these specifications are running and healthy.

kube-proxy

`kube-proxy` is a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes, which allow network communication to running Pods from network sessions inside or outside the cluster.

Container runtime

This is the component responsible for managing the execution and life-cycle of containers within the Kubernetes environment. Some examples of `container` `runtime` are containerd, CRI-O, Docker and any other implementation of the Kubernetes CRI (Container Runtime Interface).

3.3 Kubernetes objects

Kubernetes objects are persistent entities in the Kubernetes system, they are used to represent the state of the cluster. This section will provide the most important information regarding Kubernetes objects by presenting, at the same time, some of the resources that are typically used inside a Kubernetes cluster.

3.3.1 Common concepts

There are several pieces of information that are common to all the Kubernetes objects, they are:

- `apiVersion`: specifies the API version used to create the object.
- `kind`: defines the resource that this object represents.
- `metadata`: used to provide several information regarding the object, like for example name, namespace, labels and others.
- `spec`: describes the desired state of the object, this information is provided by the user.
- `status`: describes the actual state of the object, this information is provided by the server.

When creating one of these objects, the user must fill in the `spec` field that describes the desired state, furthermore additional basic information about the object (such as a name) can be provided in the `metadata` field. The object, most of the times, is described by a file known as "manifest", which is where all of its information is located. By convention, manifests are written in YAML format, but it is possible to specify them in JSON format as well. Tools such as `kubectl` convert the information from a manifest into JSON or another supported serialization format when making the API request over HTTP [13]. Whenever a new object is created, the Kubernetes control plane will continue to monitor its actual state, so that it matches the desired state specified by the user. Another important concept is the fact that it is possible to organize and mark a subset of objects, this can be done by using Labels, which are key-value pairs that can be attached to the objects, and Selectors, which are primitives that allow to select a set of objects with the same label.

3.3.2 Operations on objects

Kubernetes allows to perform different types of actions on its objects:

- **Create**: it creates the desired resource.
- **Read**: there are several variants of this operation:
  - **Get**: allows to retrieve a specific object by its name.
  - **List**: allows to retrieve a list of objects of the same type inside a namespace, additional constraints regarding the selection process can be specified through a selector query.
  - **Watch**: allows to track the stream of changes made to the object.
- **Update**: there are several variants of this operation:
  - **Patch**: changes a specific field of the object.
  - **Replace**: replaces the current `spec` with a new one.
- **Delete**: it allows to delete a specific resource.

3.3.3 Pod

Pods are are the smallest deployable units of computing that you can create and manage in Kubernetes, it is a collection of one or more relatively tightly coupled containers that share the same network and storage. Pods are ephemeral and have no auto-repair capacities, this is the reason why users never create individual Pods directly in Kubernetes, in fact they are usually managed by a controller.

3.3.4 Namespace

Namespaces provide a mechanism for isolating groups of resources within a single cluster, in fact they act as a sort of logical partition. They cannot be nested inside one another and each Kubernetes resource can only be in one namespace, as a consequence, even though names of resources need to be unique within a namespace, they can share the same name across different namespaces.

3.3.5 ReplicaSet

These are objects that allow to control a set of replica Pods running at the same time. If one of the Pods in this set is deleted, the ReplicaSet will notice that the current number of replicas is different from the desired one, as a consequence a new Pod is created to replace the deleted one. Usually ReplicaSets are not used directly, they are managed through higher-level objects called Deployments.

3.3.6 Deployment

Deployments manage the creation, update and deletion of Pods by creating a ReplicaSet at a lower level that will manage the desired number of Pods according to the logic specified in Subsection 3.3.5. An example of Deployment can be viewed by looking at Listing 3.1: this manifest creates a Deployment called `nginx-deployment` labeled as `app:nginx`. It specifies that three replicated Pods have to be created and managed, furthermore, the `selector` field specifies that this Deployment will manage Pods with label `app:nginx`. The template field is used to summarize the information of the created Pods, in fact, as it can be seen, they are labeled as `app: nginx` and will launch one container running the `nginx:1.14.2` image on port 80.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Listing 3.1: Example of Kubernetes Deployment [14]

3.3.7 Service

Services are resources used to expose network applications running as one or more Pods inside the cluster. There are several types of services, each with its own purpose depending on the situation:

- **ClusterIP**: this is the default type of Service, it is accessible only from within the cluster.
- **NodePort**: it exposes the Service on a static port of each Node's IP, this means that it can be accessed from outside the cluster.
- **LoadBalancer**: it exposes the Service externally using an external load balancer, like the cloud provider's load balancer.
- **ExternalName**: it references to an external DNS address instead of only Pods, this will allow local applications to reference external services.

An example of Service (ClusterIP) can be viewed by looking at Listing 3.2: it is named `my-service`, its purpose is to redirect requests coming from TCP port 80 to port 9376 of any Pod labeled as `app.kubernetes.io/name:MyApp`.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Listing 3.2: Example of Kubernetes Service [15]

3.4 Kubernetes scheduling concepts

There are different instruments and mechanisms that Kubernetes uses to deal with the scheduling problem, each one of them with their advantages and disadvantages. Let's introduce some of the most important ones:

3.4.1 Taints and Tolerations

They are properties, of a node and a Pod respectively, that can be used to ensure that Pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node, this marks that the node should not accept any Pods that do not tolerate (don't match) the taints, this is one of the most important advantages, as only specific Pods can be selected to run on a tainted node, thus excluding the others and ensuring a fine-grained approach for the scheduling problem inside clusters. One disadvantage is the risk of "over-tainting" nodes inside the cluster, as this can lead to scheduling issues and inefficiencies if not managed properly (for example, this can lead to pending Pods).

3.4.2 Affinity and Anti-Affinity

These are features that can be used to specify whether a Pod should be running on a dedicated node or be executed on the same node of other specific Pods (or on a totally different node if Anti-Affinity is specified). The clear advantage of these features is the considerable customization capabilities that they offer when new Pods need to be scheduled. There are two main policies that can be specified when dealing with affinities:

preferredDuringSchedulingIgnoredDuringExecution

It specifies that node affinity have to be respected when scheduling, if that is possible. If not, the Pod gets scheduled anyway according to Kubernetes default scheduling policies.

requiredDuringSchedulingIgnoredDuringExecution

It specifies that node affinity have to be respected all the time. If not, the Pod enters in pending state and cannot be executed until the affinity condition is met, this can be a disadvantage if not managed properly.

3.4.3 Kubernetes Descheduler

This isn't a native feature of Kubernetes [16], it is an additional component that can be installed inside clusters to deal with a limitation of `kube-scheduler`, Kubernetes native scheduler, which is the fact that the latter do not schedule Pods once they are scheduled the first time, this can be problematic, as flexibility can be greatly reduced when scheduling Pods. The descheduler component dynamically finds Pods that can be moved (when the appropriate conditions are met) and evicts them, several examples of this are when for example new taints are added to some nodes, new affinities are specified or removed and so on. This is a great advantage, as it automates Pods scheduling whenever necessary. It is important to note that, when Pods are evicted, the descheduler does not schedule replacement for them, but instead relies on the default scheduler (`kube-scheduler`) for that.

3.5 Prometheus

Prometheus [17] is an open-source systems monitoring and alerting toolkit. It collects and stores its metrics as time series data, this means that metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels. It offers several features, some of them are the following:

- It offers a multi-dimensional data model (time series defined by metric name and set of key/value dimensions).
- It provides a powerful and flexible query language to leverage this dimensionality called PromQL.
- Uses an HTTP pull model for time series collection.
- Targets are discovered via service discovery or static configuration.
- Multiple modes of graphing and dashboarding support.

3.5.1 Prometheus architecture

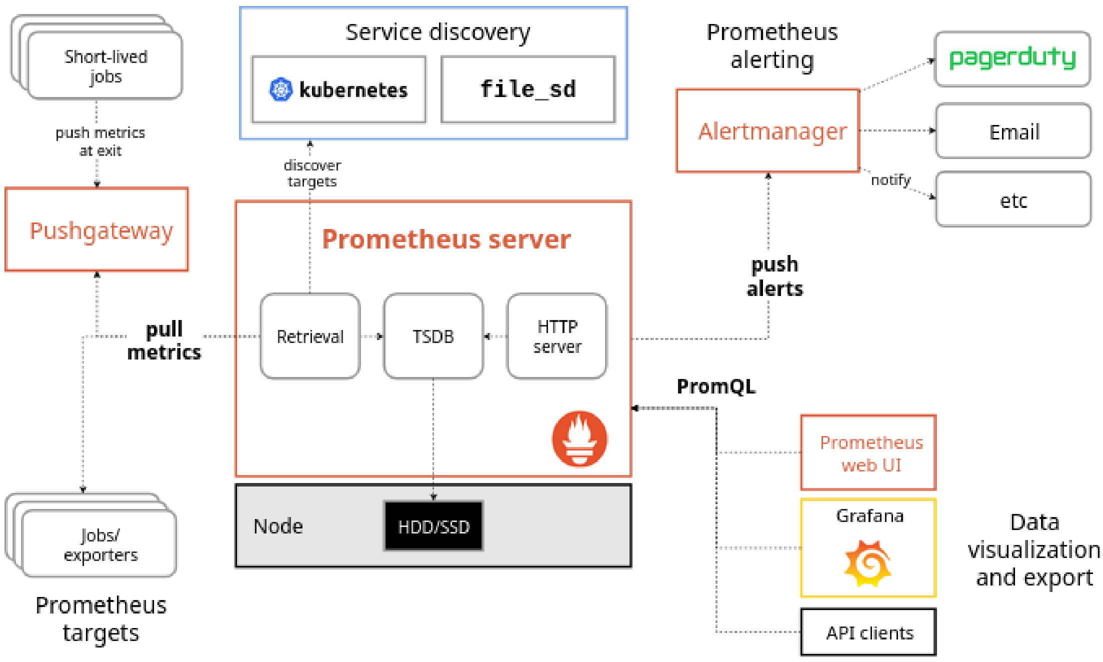


Figure 3.2: Prometheus architecture [18]

The architecture of Prometheus can be viewed by looking at Figure 3.2, let us analyze some of its most important components:

Prometheus server

This component is formed by three parts:

- **Retrieval**: it is responsible for collecting metrics from the targets through HTTP requests, which will be stored in a time series database.
- **TSDB**: this database is where all the metrics are stored, its designed to support the requirements of monitoring by handling large-scale, high-throughput metrics collections by providing at the same time fast and reliable access to historical data.
- **HTTP server**: it is the component responsible of retrieving the data stored in the database, this can be done by sending to this server a PromQL query for the desired metrics.

Pushgateway

Even though Prometheus follows a pull-based model, the Pushgateway has been introduced in order to deal with scenarios where metrics need to be pushed by short-lived batch jobs that cannot be scraped. This component can act as a temporary buffer that accepts pushed metrics from these jobs, thus making them available for scraping by the Prometheus Server itself.

Alertmanager

This component handles alerts generated by Prometheus, it allows for routing, grouping, and processing of alerts that can be sent to various notification channels. Examples of these channels are the email, Slack, and others.

Exporters

These components extract metrics from the target system, converts them into the Prometheus data format and expose an HTTP endpoint where the Prometheus Server can scrape the metrics. There are numerous third-party exporters available for popular systems and frameworks, otherwise one can create its own custom exporters to expose specific metrics. Conventionally, metrics are exposed on the `/metrics` endpoint of each target, but this can be changed by configuring Prometheus.

Service Discovery

This process allows Prometheus to automatically discover and monitor targets without any manual configuration.

Grafana

While not a core component of Prometheus itself, Grafana [19] is commonly used to display data coming from various data source such as Prometheus, it is a popular open-source analytics and monitoring platform that provides a user-friendly interface to create custom dashboards and graphs.

3.5.2 Prometheus metrics

There are four types of metrics that Prometheus offers:

- **Counter**: this is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart.
- **Gauge**: it represents a single numerical value that can arbitrarily go up and down.
- **Histogram**: it samples observations and counts them in configurable buckets. It also provides a sum of all observed values.
- **Summary**: it samples observations; while it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

3.5.3 Discovering Kubernetes targets with Prometheus

Some projects make it possible to easily integrate Kubernetes and Prometheus, the most famous one is probably Prometheus Operator [20], as it provides Kubernetes native deployment and management of Prometheus and related monitoring components; the purpose of this project is to simplify and automate the configuration of a Prometheus based monitoring stack for Kubernetes clusters, furthermore it uses Kubernetes custom resources to deploy and manage Prometheus, Alertmanager, and related components. Sometimes it is convenient to generate metrics from applications exposed by Kubernetes Services, just like if they were regular Prometheus targets; for this reason Prometheus Operator provides the `ServiceMonitor` CRD, which is a resource used to identify Services that are used to expose applications that generate metrics to scrape from within Kubernetes, additionally it specifies how often these metrics should be scraped. An example of ServiceMonitor can be viewed by looking at Listing 3.3: it specifies a ServiceMonitor resource called `prometheus-k8s` inside the `monitoring` namespace, furthermore several labels are associated to it; scrapings should be performed every 30 seconds on Services associated to all the labels present in the `spec.selector.matchLabels` field.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 3.4.0
  name: prometheus-k8s
  namespace: monitoring
spec:
  endpoints:
  - interval: 30s
    port: web
  - interval: 30s
    port: reloader-web
  selector:
    matchLabels:
      app.kubernetes.io/component: prometheus
      app.kubernetes.io/instance: k8s
      app.kubernetes.io/name: prometheus
      app.kubernetes.io/part-of: kube-prometheus
```

Listing 3.3: Example of ServiceMonitor [21]

Chapter 4

Liqo

This chapter will introduce Liqo by discussing its main motivations and concepts. It is worth mentioning that this chapter will specifically refer to Liqo v0.10.3, the version used during the development of this work.

4.1 Liqo: motivations

Liqo [22] is an open-source project that enables dynamic and seamless Kubernetes multi-cluster topologies, supporting heterogeneous on-premise, cloud and edge infrastructures. It aims at connecting Kubernetes clusters together, so that they may work alongside each other to reach shared goals; this is particularly important because, sometimes, these clusters tend to have more resources than needed, as a consequence it would be beneficial for other clusters to use some of these unused resources when they are overloaded. All of this is made possible by the fact that Liqo is capable of establishing peering sessions between these clusters, thus creating a larger virtual cluster that hosts the sum of the resources exposed by each cluster involved in the peering process; everything is kept simple, as clusters see their peers just as **virtual nodes** that are automatically added to the group of its other (real) nodes, this allows to schedule some of the cluster's task to these virtual nodes as well. By scheduling parts of its workload to the other clusters, all of the unused resources that would normally be wasted are now leveraged in a smart and convenient way, hence multi-cluster topologies are now possible and beneficial.

4.2 Liqo main concepts

This section will discuss some of the most import foundations behind Liqo, specifically it will just focus on the theoretical aspects behind tasks' offloading that were particularly relevant in this work.

4.2.1 Liqo peering

In Liqo, the term peering indicates a unidirectional resource and service consumption relationship between two Kubernetes clusters, where one consumer cluster has the capability to offload tasks to a provider cluster, but not vice versa. To be specific, the consumer establishes an outgoing peering towards the provider, which in turn is subjected to an incoming peering from the consumer; of course, by combining two simultaneous peerings, one could achieve a bidirectional peering where both of the clusters are consumer and provider at the same time. Let us discuss in more details how the peering process works [23]:

- **Authentication**: each cluster, once properly authenticated through pre-shared tokens, obtains an identity to interact with the other cluster, which will be used to negotiate the necessary parameters in the next phase of the peering process.

- **Parameters negotiation**: the two clusters involved in the peering process exchange the set of parameters required to complete the peering establishment; examples of parameters are the amount of resources shared with the consumer cluster, the information concerning the setup of the network VPN tunnel, and others.

- **Virtual node setup**: this is the point where the consumer cluster creates a new virtual node for the provider cluster, thus abstracting the resources shared by the latter.

- **Network fabric setup**: the two clusters configure their network fabric and establish a secure cross-cluster VPN tunnel, according to the parameters previously negotiated. The network fabric is a subsystem that transparently extends the Kubernetes network model across multiple independent clusters, this enables Pods hosted by the local cluster to seamlessly communicate with the Pods offloaded to the remote one, regardless of the underlying CNI plugin and configuration.

There are two non-mutually exclusive peering approaches that could be used between two clusters:

- **Out-of-band control plane**: by choosing this strategy, which is the default one, Liqo control plane traffic will flow outside the VPN tunnel interconnecting the two clusters; additionally, it is characterized by high dynamism, as upon parameters modifications the negotiation process ensures synchronization between clusters and the peering automatically re-converges to a stable status.

- **In-band control plane**: in this approach, Liqo control plane traffic will flow inside the VPN tunnel interconnecting the two clusters, as a consequence, if parameters are modified, the connectivity is lost due to the fact that the cross-cluster VPN tunnel was statically configured, hence manual intervention is required.

4.2.2 Liqo offloading

In order to enable seamless workload offloading, Liqo extends Kubernetes namespaces across the local cluster boundaries, this means that, once a given namespace is selected for offloading, Liqo will proceed with the creation of a twin namespace in the remote cluster. Remote namespaces host the actual Pods offloaded to the corresponding cluster, as well as other additional resources that are present in the local namespace, which are propagated by the resource reflection process. As can be seen by looking at Figure 4.1, there is a given namespace existing in the local cluster and extended to a remote one; a group of Pods is contained in the local namespace, while a subset is scheduled onto the virtual node and offloaded to the remote namespace (they are the faded-out ones). Furthermore, as already specified before, the resource reflection process propagated other resources in the remote namespace, this is done in order to make sure that Pods are properly executed. The main advantage of this process is that it provides many possibilities regarding Pods' placement, for example it is possible to choose whether to schedule Pods onto physical nodes, virtual ones or both, another example is that users could select a specific subset of the available remote clusters by means of standard selectors matching the label assigned to the virtual nodes, so that the workload may be offloaded only to specific clusters. When a Pod is scheduled onto a virtual node, a twin Pod object is created in the remote cluster for actual execution. It's worth mentioning that Liqo provides 3 different offloading strategies for namespaces:

1. **LocalAndRemote**: Pods deployed in the local namespace can be scheduled both onto local nodes and onto virtual nodes, hence possibly offloaded to remote clusters; this is the default offloading strategy.

2. **Local**: Pods deployed in the local namespace are enforced to be scheduled onto local nodes only, hence never offloaded to remote clusters.

3. **Remote**: Pods deployed in the local namespace are enforced to be scheduled onto remote nodes only, hence always offloaded to remote clusters.

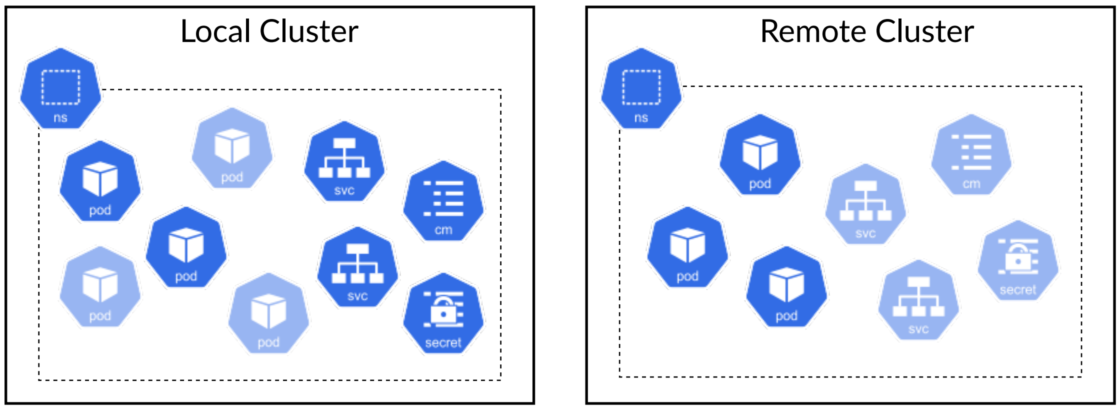


Figure 4.1: Namespace offloading in Liqo [24]

Chapter 5

FLUIDOS

This chapter will present the FLUIDOS project by discussing its goals and how it can provide a transparent computing continuum. As a conclusion the DEAS sub-project, which is what this thesis work has focused on, will be presented as well.

5.1 FLUIDOS: introduction

The "**Flexible, scaLable, secUre, and decentralIseD Operating System**" (FLUIDOS) [25] aims to leverage the enormous, unused processing capacity at the edge, scattered across heterogeneous edge devices that struggle to integrate with each other and to coherently form a seamless computing continuum. As a matter of fact, the constant growing in the adoption of the edge computing paradigm has inevitably led to an increase of the computational capabilities located at the edge of the network, as a consequence it would be convenient to leverage it by creating a transparent computing continuum that can efficiently move tasks between different layers of the network, just as already explained in Section 1.1. This the reason behind the creation of FLUIDOS, in fact its purpose is to create a computing infrastructure that allows to unify all the different layers of the network, thus making the physical location of where tasks are executed irrelevant. Considering that FLUIDOS uses Kubernetes's capabilities in order to work efficiently in cloud-native environments, the goal is to create just one "virtual" cluster coming from the union of multiple Kubernetes clusters where applications can be executed and moved from one cluster to the other seamlessly, so, for this other reason, FLUIDOS uses Liqo functionalities as well in order to create peerings between clusters and offload the workload as needed.

5.2 FLUIDOS: a transparent computing continuum

FLUIDOS is capable of creating a virtual computing environment that can seamlessly offload tasks between clusters, hence creating a transparent computing continuum that can leverage efficiently the unused computational power present at the edge of the network. To be specific, in FLUIDOS, Kubernetes clusters are typically referred as **FLUIDOS Nodes**, even though they are much more than that. This section will present what a FLUIDOS Node is and what allows them to discover other Nodes.

5.2.1 FLUIDOS Node

A FLUIDOS Node, as already discussed, essentially corresponds to a Kubernetes cluster. This Node is orchestrated by a single control plane instance, and it can be composed of either a single machine, like an embedded device, or a set of servers, like a datacenter. Of course, device homogeneity is desirable, as it allows to simplify the overall management, but it is not requested within a FLUIDOS Node, this allows to integrate heterogeneous devices between them. A FLUIDOS Node handles problems such as orchestrating computing, storage, network resources and software services within the cluster and, by leveraging Liqo's capabilities, can transparently access to resources and services that are running in another remote cluster, which is in fact a remote FLUIDOS Node. Among its main components, there is one that is particularly relevant for the service provisioning process, which is the The REAR protocol.

REAR

The "**REsource Advertisement and Reservation**" (REAR) protocol [26] enables different actors such as cloud providers and customers to advertise and reserve resources and/or services. Specifically, this is the protocol that allows FLUIDOS Nodes to exchange information about their available resources and services with other Nodes; this is convenient, as it allows them to filter remote Nodes depending on whether the resources and services they can offer match the requirements of the local Node.

5.2.2 Discovery of new Nodes

An important concept in FLUIDOS is the **Network Manager**, which is a component that implements a multicast-based discovery protocol that allows FLUIDOS Nodes to learn about nearby Nodes, so that they may establish new peerings. For each Node, there will be a `KnownCluster` CRD, which is a resource that contains all the relevant information regarding that specific FLUIDOS Node. There is one important limitation though, which is the fact that multicast transmissions are restricted on the Internet, so, for this reason, the Network Manager can only work properly in private networks. In order to address this problem, FLUIDOS later introduced **Neuropil** [27], a discovery protocol that allows to find other FLUIDOS nodes on the Internet, this is done by creating an overlay network where multicast transmissions are not used; it is worth mentioning that, during the development of this work, this last feature was not yet available, hence an alternative strategy had to be conceived (more details in Section 6.5).

5.3 DEAS

The "**Distributed Edge Analytics Service**" (DEAS) open call is a FLUIDOS sub-project that focuses on enhancing the capabilities of the latter by providing the appropriate support for seamless operations even in fluctuating network conditions, like in 5G networks, thus improving the already existing computing continuum by making it responsive and adaptive. The main goals of the project can be summarized as follows [28]:

- **Real-Time Data Processing**: Enable real-time data processing and analytics at the network's edge.
- **Resource Optimization**: Provide telemetry and radio link information for optimal resource allocation.
- **Service Continuity**: Enhance service and computing continuity in dynamic environments, such as drone swarming.
- **Dynamic Resource Distribution**: Facilitate dynamic distribution of computational resources among micro edges.

5.3.1 Addressing fluctuating network conditions

Considering that the 5G network can be unreliable and that the bandwidth between the clusters may be insufficient, DEAS will use 5G small cells to receive real-time channel metrics that can be leveraged to have an indication about the quality of the channel itself. These metrics will be used according to the following logic: if the channel quality is good enough, then all the workload will be offloaded from the drones to the network's edge, on the contrary, if the quality is bad, the workload will return to the drones (Figure 5.1). The channel quality management process allows to improve the overall resource usage, thus overcoming the current limitations regarding the drones' battery and hardware by lightening their workload when possible, and, at the same time, leveraging the unused computational power present at the network's edge to execute drones' tasks. Another clear advantage is the fact that there is no need for any manual configuration regarding tasks' allocation, as everything is done automatically depending on the network's condition.

5.3.2 Channel quality representation

The process of estimating the quality of the 5G channel by looking at the metrics offered by the 5G cell can be quite complicated, as many different metrics need to be considered for our specific use case. To simplify the management of the channel quality, it has been decided to use as a reference not all the metrics taken individually, but rather a single value that "summarizes" them, which, in the context of this project, is called "score value". This score value, just as all the other metrics, is offered by the 5G cell and the higher it is the more the channel quality is good. The score will be used according to the logic already specified in Subsection 5.3.1.

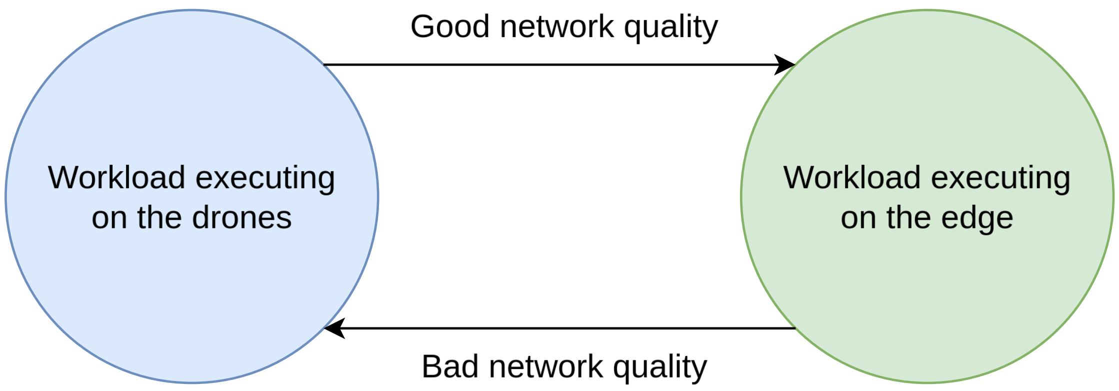

Good network quality
Workload executing on the drones
Workload executing on the edge
Bad network quality

Figure 5.1: DEAS offloading logic

Chapter 6

Edge offloading for drone swarming scenarios

This chapter will discuss the design and implementation of the system required to achieve an autonomous, adaptive, and efficient workload distribution, where tasks are automatically moved from the drones to the network's edge and vice versa by using a 5G network. The first topic that will be covered is how the score value fetched from the 5G cell can be used to decide if it is the case to move the drones' workload or not, after that, two possible design solutions for our system will be taken into account by examining the advantages and disadvantages of each one of them. Lastly, the architecture of the system will be shown and discussed by examining every single component and its purpose.

6.1 Score management and task offloading

Considering that the offloading operation requires the temporary termination of the tasks, an eventual fluctuation of the score value would cause several interruptions of the service. In order to deal with this problem, it would be better to take as reference not a single score value, but instead an average calculated over a range of N score values as follows:

$$Average\ score = \frac{1}{N}\sum_{i=1}^{N} Single\ score\ value_i \tag{6.1}$$

From now on, for the sake of simplicity, this thesis will refer to the average score just as "score", "score value" or "channel score". Now let us talk about how the system can understand when the score value is high or low enough to perform an offloading operation by introducing two important concepts that play a role in this scenario:

- **Score Threshold**: this value will be used to distinguish high score values from low score values.
- **Delta**: Given the fact that the score fluctuation phenomenon still exists, a delta value is needed in order to deal with it.

The logic is the following: whenever we are in the situation depicted by Equation 6.2, no action will be performed since the score has not changed enough to justify an offloading operation, this will allow the system to deal with the aforementioned score fluctuations.

$$Threshold - \frac{Delta}{2} \leq Score \leq Threshold + \frac{Delta}{2} \tag{6.2}$$

Let us see a real example by looking at Figure 6.1, where the evolution of the score over time is graphically represented. In this example, the score threshold is set to 1.5 (the yellow line), while the delta is set to 1, this means that whenever the score value is located between 1 and 2 (the green and red lines respectively) the system will perform no action. Whenever the score is less than 1, a local offloading will be performed, so the tasks will be moved from the edge to the drone, on the contrary, whenever the score is greater than 2, a remote offloading has to be performed in order to move the tasks' execution from the drone to the edge.

6.2 System design and functioning

This section will present the architecture of the system by illustrating its overall functioning and the main tools that were used to develop it.

6.2.1 Architecture of the system

The architecture of the offloading system is represented in Figure 6.2, for the sake of simplicity this diagram takes into account the case of just a single drone interacting with the network's edge, even though multiple drones can perfectly interact with the edge in parallel, further details regarding this aspect will be shown later. As it can be seen, the diagram depicts two FLUIDOS Nodes, the local one (the drone) and the remote one (the network's edge), the latter will be responsible of fetching the channel score from an external metrics server, which is located on the 5G cell. The HTTP calls to the external server are periodically performed by a custom **Prometheus exporter** that provides this score value as a Prometheus metric. At this point, the score can be used by another component that is located inside the drone called **Arbiter**, which is responsible of the offloading operations

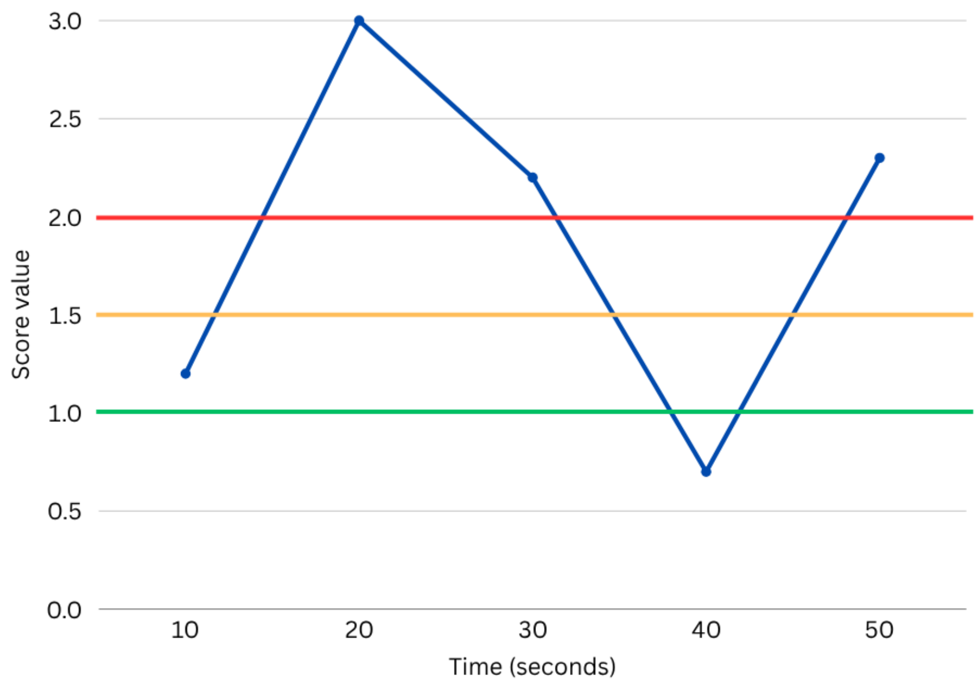


Figure 6.1: Example of score evolution over time

themselves[1], the workload in this case is represented by a single task that can be moved from the drone to the edge and vice versa, depending on the situation. Another important aspect to take into account is the fact that the drone and the edge could be separated by multiple network devices, this means that at times the drone could not be able to establish a peering with the remote cluster by using FLUIDOS's default features, which, during the development of this work, were not yet able to perform this operation; for this reason an additional component that can discover new clusters that are not directly connected to the drone has been created, which is called **Discovery Agent**. Now let us present the two main cases that could happen in this scenario, for this discussion we will refer to the same threshold and delta values that were used in Section 6.1, which are 1.5 and 1 respectively. The first case is the one where the channel score is not high enough, let us say

[1]The Arbiter uses the channel score to understand where the workload should be offloaded for execution; at that point, this component will perform the offloading operation.

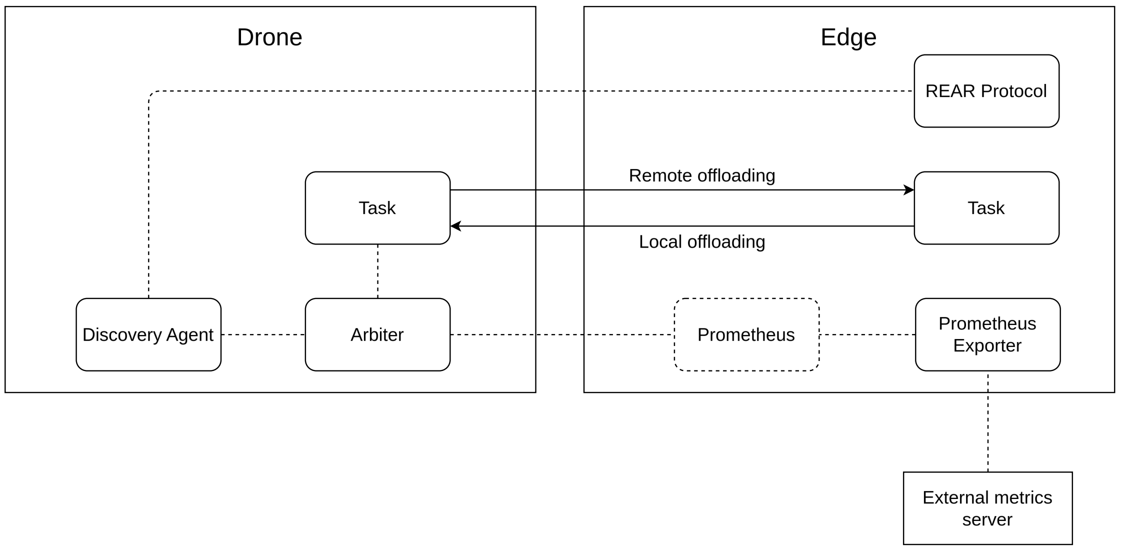


Figure 6.2: System architecture diagram

for example that it has a value of 0.5, what would happen is the situation that is depicted in Figure 6.3, where the workload is located on the drone for execution. The other case is the one where the channel score is high, let us say 5. As can be seen in Figure 6.4, the workload is placed on the network's edge for execution.

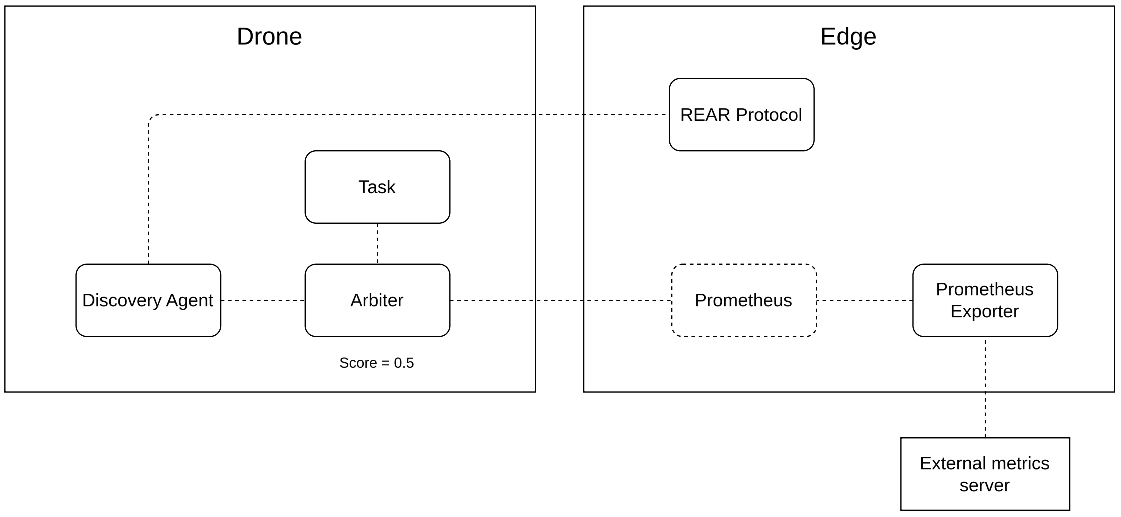


Figure 6.3: Bad score case example

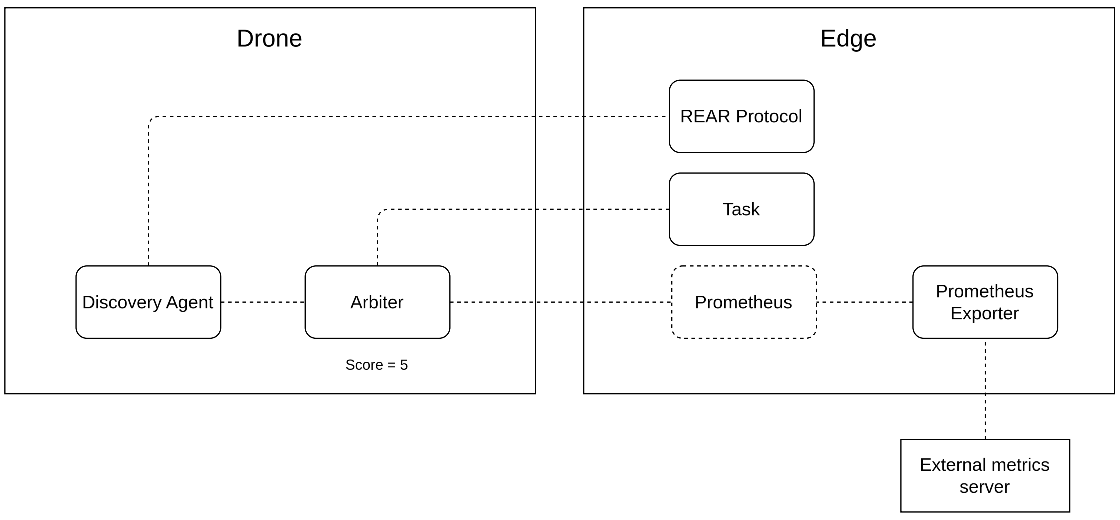


Figure 6.4: Good score case example

6.2.2 Technical design choices

The system uses Prometheus for storing score values as time series, this is a convenient representation for numerical metrics that change over time, furthermore Prometheus exporters are particularly useful when there is need to translate some data fetched from an external source to a syntax that Prometheus can understand, this happens for example in this scenario, where the data returned by the external metrics server is represented by using JSON. Every single component present in the architecture has been written by using the Golang language, one of the reasons is that its the most used language when it comes to Cloud native applications, as all major Cloud providers use Go APIs for their services, but most importantly, it integrates perfectly with all the main technologies used in this field, such as Kubernetes and Docker, which are both written in Golang [29]. Another reason is that Golang provides a powerful set of tools for interacting with Prometheus in a convenient way, such as the Prometheus Go client library[2]. The system uses Liqo v0.10.3 and FLUIDOS v0.1.1 APIs to interact with their respective resources.

[2]https://github.com/prometheus/client_golang

6.3 Possible design solutions for the offloading process

There are two possible solutions that could be alternatively used to implement the offloading logic, each one with their advantages and disadvantages.

6.3.1 First solution

This solution completely relies on Kubernetes native features, so there is no need to complicate the system architecture by adding new components. Let us discuss about the local and remote offloading operations by analyzing the actions that are required to implement the both of them.

- **Local offloading**: in order to perform this kind of operation, the arbiter needs to mark the virtual node representing the remote cluster with an additional taint that allows the Kubernetes scheduler to evict the tasks that were previously offloaded to it. At the same time, the arbiter has to change Liqo's offloading strategy to "Local" on all the namespaces that contain the offloaded tasks, by doing so it can be assured that local execution is enforced.
- **Remote offloading**: the remote offloading operation has to be performed by removing the previously mentioned additional taint (if present), so that the tasks can be chosen for execution onto the virtual node without restrictions, furthermore, the namespaces' offloading strategy has to be set to "Remote". The inconvenient is that, by doing so, the tasks aren't actually "forced" to execute onto the virtual node, in order to do that the Pods representing the tasks have to be manually deleted and recreated, this guarantees that the changes will be applied to the new Pods.

In this case, the architecture of the system will remain the same as the one that was already presented in Section 6.2, as can be seen in Figure 6.5.

6.3.2 Second solution

This solution exploits the Kubernetes Descheduler as an additional component to compensate the fact that the first one requires to manually delete the Pods in order to apply new scheduling policies. The first thing is to install the Descheduler inside the local cluster and configure it with the `RemovePodsViolatingNodeAffinity` policy (which will be able to make sure that affinity properties are automatically applied, even when Pods are already running), the type that has to be specified is `preferredDuringSchedulingIgnoredDuringExecution`; after that, it is needed

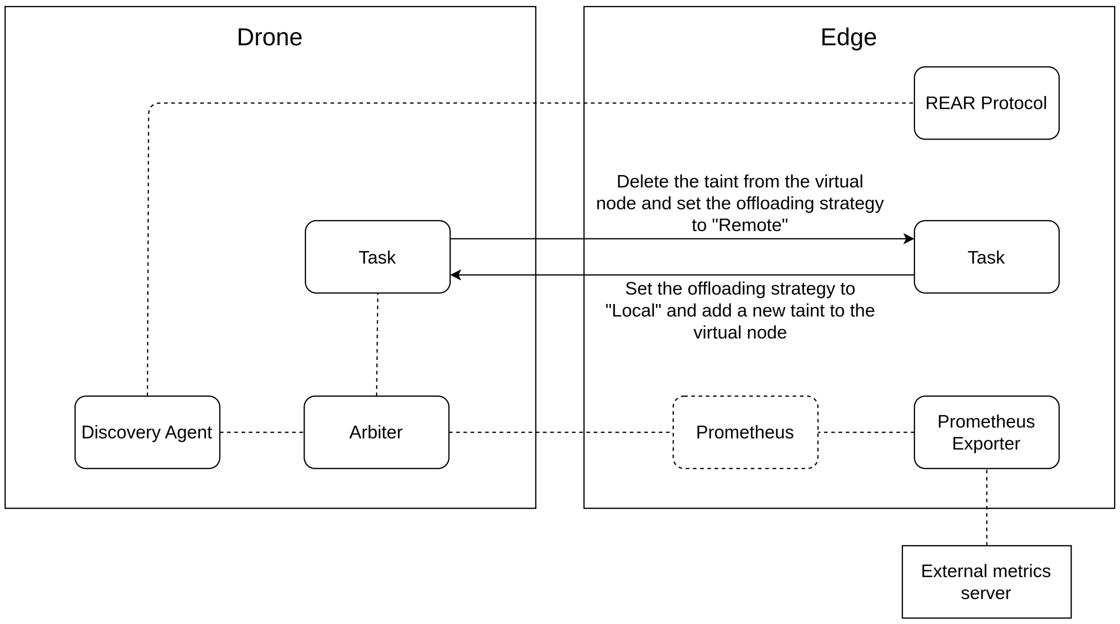

Drone
Edge
REAR Protocol
Delete the taint from the virtual node and set the offloading strategy to "Remote"
Task
Task
Set the offloading strategy to "Local" and add a new taint to the virtual node
Discovery Agent
Arbiter
Prometheus
Prometheus Exporter
External metrics server

Figure 6.5: First solution architecture diagram

to define an affinity property for all the tasks[3] that can leverage the benefits offered by the Descheduler. At this point, a new label has to be added to the virtual node, so that it can be used for the tasks' affinity; all of this is needed because the affinity mechanism will be used to perform the remote offloading operation by suggesting a better node for execution (the virtual node) to the Kubernetes scheduler, further details will be provided later. In addition to that, a taint property will also be used in order to perform the local offloading by specifying that the tasks can no longer execute on the virtual node. The steps are the following:

- **Local offloading**: the logic is the following, the label used to mark the virtual node, if present, has to be removed, so that the Kubernetes scheduler won't be able to prefer the virtual node over the other ones; after this is done, a new taint has to be added to the virtual node, thus preventing the tasks from executing on it. This operation requires to pay particular attention in executing the steps exactly as presented, because considering that the taint has to be used in conjunction with an affinity property, the offloading logic could behave not quite as expected, for example, if the taint is added before removing the affinity label from the virtual node, the Kubernetes scheduler will give priority to the affinity by trying to execute the tasks on the virtual

[3]With the same type, `preferredDuringSchedulingIgnoredDuringExecution`.

node, this won't be possible since a new taint has been previously specified, the result is that the tasks will constantly be terminated and restarted on the local cluster since execution is forbidden on the remote one under those specific conditions.

- **Remote offloading**: the logic here is the exact opposite of the previous case, so the virtual node taint, if present, has to be removed to allow execution; after that, a label has to be specified for the virtual node in order to make sure that the Kubernetes scheduler will prefer that specific node for tasks' execution.

The introduction of the Descheduler slightly modifies the architecture, as can be seen in Figure 6.6.

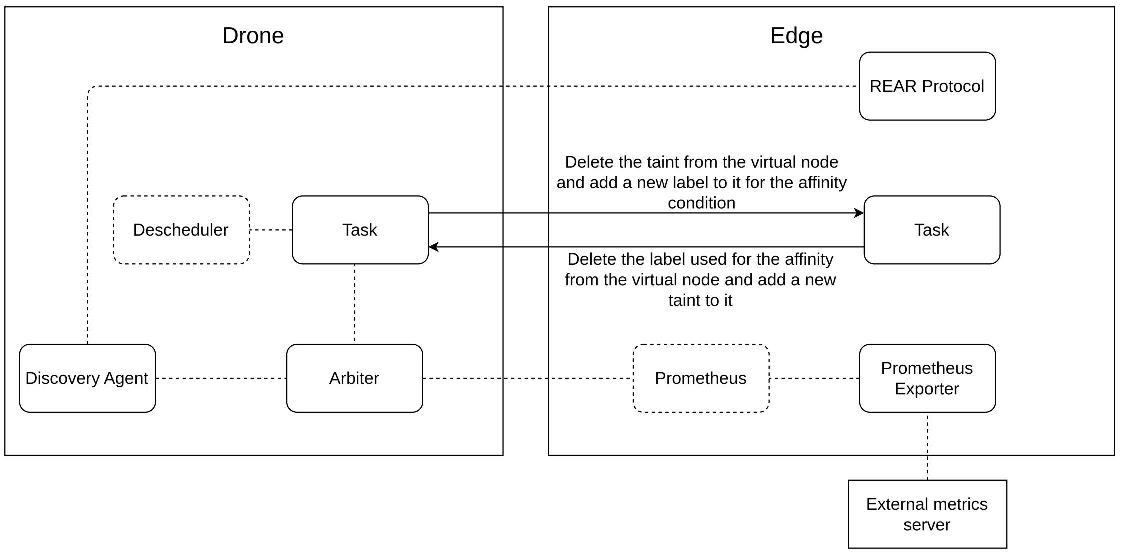

Drone
Edge
REAR Protocol
Delete the taint from the virtual node and add a new label to it for the affinity condition
Descheduler
Task
Task
Delete the label used for the affinity from the virtual node and add a new taint to it
Discovery Agent
Arbiter
Prometheus
Prometheus Exporter
External metrics server

Figure 6.6: Second solution architecture diagram

6.3.3 Confronting the two solutions

Now let us compare the two possible solutions that were previously discussed by presenting their advantages and disadvantages.

First solution

- Advantages:

  1. This solution completely relies on Kubernetes native features, there's no need to install additional components.

2. The offloading logic is kept simple.
3. There's no need to use complex scheduling mechanisms.

- Disadvantages:
  1. When executing a remote offloading, in order to apply the changes[4], the Pods have to be manually deleted and recreated.
  2. The offloading strategy has to be changed every time an offloading operation occurs.

Second solution

- Advantages:
  1. This solution allows to dynamically schedule Pods whenever necessary, even during their execution, as a consequence there's no need for any manual intervention[5].
  2. There's no need to change Liqo's offloading strategy.
- Disadvantages:
  1. The Descheduler needs to be installed and configured in order to implement this solution.
  2. An affinity property has to be specified for the task, furthermore a label needs to be added to/removed from the virtual node at every offloading operation.

Final decision

By confronting the two solution, one can note that it is preferable to not rely on additional components that have to be manually installed on each drone, such as the Kubernetes Descheduler, when there is an alternative that is completely based on Kubernetes native features; furthermore, considering that a controller that modifies various scheduling properties has to be created anyway (the Arbiter), there is no point in adding a Descheduler just to automatize the descheduling of the Pods, this would create an additional useless dependency. Considering all the aforementioned reasons, the first solution is preferable for the specific use case of this project, so it is going to be used to implement the offloading process logic.

[4]Taint deletion and offloading strategy set to "`Remote`".

[5]No need to manually delete Pods.

6.4 Exporting the channel score to Prometheus

This section will explain how the system fetches the required channel scores and how it makes them available as Prometheus metrics. As a conclusion, a possible alternative to the channel score will be presented, one that does not require to contact an external metrics server by going through the network, but instead relies on an approximative value of the channel quality, which is the 5G signal strength.

6.4.1 Interacting with the external metrics server

The metrics server has 3 endpoints, each one of them has to be contacted in order to get the channel scores, this is because, inside the path relative to the endpoint responsible for providing said scores, two pieces of information that can only be returned by the other two endpoints have to be specified; they are the so called `gnb_id`, which identifies the gNB device, and the `imsi` of all the drones. The endpoints are:

1. `/amf_gnb_stats`: This endpoint returns an array of JSON objects represented in Listing A.1; the only relevant piece of information is the third field of every object, which is the `gnb_id`. There is only one gNB in this scenario, so there will always be just one entry in the returned array.

2. `/amf_ue_stats`: This endpoint returns an array of JSON objects represented in Listing A.2; the only relevant piece of information is the seventh field of every object, which is the `imsi` value of a single drone. Considering that there could be multiple drones connected to the 5G cell, it is more likely that this array will contain more entries, one for each drone.

3. `/ue_mac_stats/{gnb_id}/imsi/{imsi}/score`: This endpoint returns the JSON object represented in Listing A.3; depending on the `imsi` value used, the channel score of the connection between the 5G cell and that specific drone will be returned, which is the second field of the response.

6.4.2 Collecting the metrics for Prometheus

The component that will contact the metrics server is a custom Prometheus exporter located inside the edge cluster. Let us analyze its functioning by looking at Listing 6.1; the first action is to get two parameters that are necessary for the exporter, the first one is the URL of the API server, which is the aforementioned metrics server, the other one is the port on which the exporter will listen for new requests. The next three instructions create the so called `ScoreCollector` struct (Listing A.4), which implements the `Collector` interface in order to collect new score values, and register it. The last passages are to make sure that the exporter can handle incoming requests to the endpoint `/metrics`, which is where the score metrics are available, and make the exporter listen for new requests on the port that was previously passed as a configuration parameter.

```go
func main() {

    var apiServerUrl, exporterPort = normalizeConfigurationParameters
  ↪ (os.Getenv("API_SERVER_URL"), os.Getenv("EXPORTER_PORT"))

    scoreCollector := scorecollector.NewScoresCollector(apiServerUrl)

    reg := prometheus.NewRegistry()

    reg.MustRegister(scoreCollector)

    // Start the HTTP server on the desired port.
    slog.Info("Starting server on port " + exporterPort)
    http.Handle("/metrics", promhttp.HandlerFor(reg, promhttp.
  ↪ HandlerOpts{}))
    http.ListenAndServe(":"+exporterPort, nil)
}
```

Listing 6.1: Main function for the Prometheus exporter

Every single request that the exporter receives is going to trigger the `Collect` function of the `ScoreCollector` struct, its logic can be seen by looking at Listing 6.2. The first operation it executes is to call the function `getScores`, which takes as a parameter the URL of the API server (the metrics server) to contact the three endpoints of said server and returns a map of IMSIs associated to their relative score values called `ScoreMap` (Listing A.4); more details about this function will be provided later. After this is done and new score values are retrieved, if something went wrong the exporter signals it by sending back to Prometheus an HTTP 500 error, just as expected from every Prometheus exporter [30], otherwise a new metric will be created for every single entry of the map returned by `getScores`, where the IMSI and the channel score of the drone to which the current entry is referring to are used as the label and the value of the metric respectively.

```go
func (collector ScoreCollector) Collect(ch chan<- prometheus.Metric)
  ↪ {

    var scores, err = getScores(collector.apiServerUrl)

    if err != nil {
        ch <- prometheus.NewInvalidMetric(
            prometheus.NewDesc("fluidos_5g_channel_score_error",
                "An error occured while getting new channel scores.",
    ↪ nil, nil),
            err)
    } else {

        for imsi, singleScore := range scores {
            ch <- prometheus.MustNewConstMetric(collector.ScoreValue,
    ↪ prometheus.GaugeValue, singleScore, imsi)
        }
    }
}
```

Listing 6.2: Implementation of the `Collect` function

6.4.3 Executing the HTTP calls to the endpoints

Now let us analyze in further details the `getScores` function, this is where the actual HTTP requests to the endpoints are sent. The first call is made to the endpoint `/amf_gnb_stats` in order to get the gNB ID, this request can be seen by looking at Listing 6.3, where the function `getContent` is just a utility function that has been created to act as a wrapper to the HTTP call itself by returning its body content in string format. As it can be seen, we just get the first entry, as our system has just a single gNB.

```
bodyContent, err := getContent(apiServerUrl + "/amf_gnb_stats")

if err != nil {
    return nil, err
}

err = json.Unmarshal([]byte(bodyContent), &gnb_array)

if err != nil {
    return nil, err
}

if len(gnb_array.Series[0].Values) == 0 {
    return nil, errors.New("unable to get the score: no gNBs
    ↪ available")
}

// For the sake of simplicity, since we have a single gnb we just get
    ↪ the first result
```

```
gnbIDFirstRow := gnb_array.Series[0].Values[0]
```

Listing 6.3: Getting the gNB ID

The next request is sent to the endpoint `/amf_ue_stats` in order to get all the IMSIs of all the drones connected to the 5G cell, the procedure is similar to the first call, the only difference is that, in this case, we consider the entire array, since all the IMSIs are required. The last thing to do is to get all the score values for every IMSI that the second endpoint returned, this can be seen by looking at Listing 6.4. For every single IMSI (drone), the program executes a call to the third endpoint by using the gNB ID returned by the first endpoint and the IMSI value of the drone that we are referring to in the current cycle of the for loop. After the HTTP call has been performed, the IMSI value is associated to its relative score value by using a `ScoreMap` variable called `newScores`, which is later returned to the `Collect` function.

```
for i := 0; i < len(ue_array.Series[0].Values); i++ {

    ueFirstRow := ue_array.Series[0].Values[i]
    gnbID := gnbIDFirstRow[2].(string)

    bodyContent, err = getContent(
        apiServerUrl + "/ue_mac_stats/" +
            forgeGnbID(gnbID) + "/imsi/" +
            ueFirstRow[6].(string) + "/score")

    if err != nil {
        return nil, err
    }

    err = json.Unmarshal([]byte(bodyContent), &score_array)

    if err == nil {
        newScores[ueFirstRow[6].(string)] = float64(score_array.Score
    ↪ )
    }
}

return newScores, err
```

Listing 6.4: Getting the score values for all the IMSIs

6.4.4 Possible alternative to the channel score: the 5G signal strength

Even though this project uses the channel score just as presented in this section, there is a problem, which is the fact that all the channel scores are located on an external metrics server, this could lead to a small degradation in performances, as the network needs to be traversed multiple times. In this thesis work, a potential alternative to the channel score has been conceived, it consists in using an indicator of the channel quality that the drone can get locally, this is the 5G signal strength that is present between the drone itself and the 5G cell. The resulting architecture can be seen by looking at Figure 6.7, in this case Prometheus is going to run in the local cluster alongside an additional component that replaces the Prometheus exporter, which is called **dB exporter**, as its purpose is to fetch the value in decibels of the 5G signal strength that is locally available in order to expose it as a Prometheus metric. Its worth mentioning that this is only an alternative design proposition, no real implementation was done for this part, its logic has only been simulated for testing purposes. More details about this subject will be provided in Subsection 7.4.

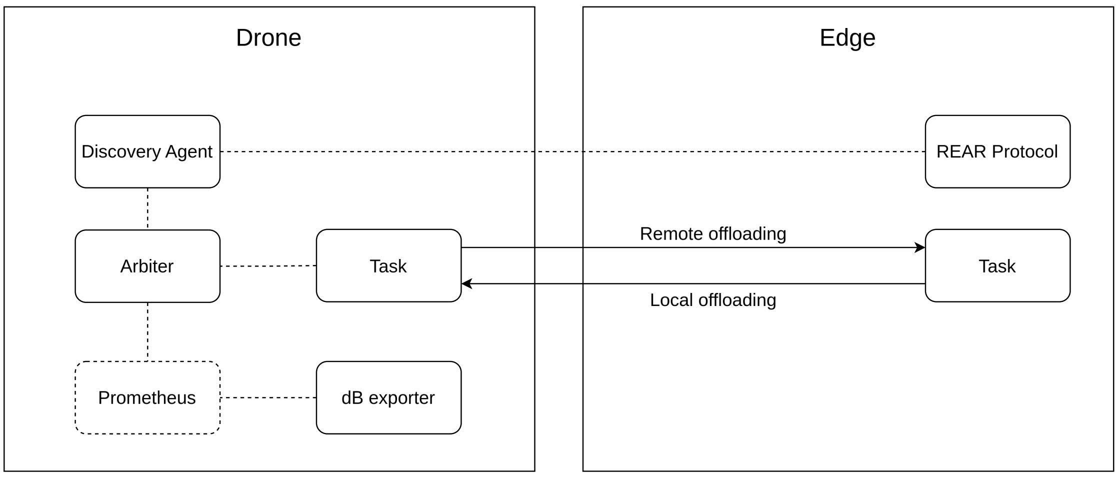


Figure 6.7: System architecture where the 5G signal strength is used

6.5 Discovery of new clusters

As already mentioned in Section 6.2, the drones and the network's edge could be separated by multiple network devices; this can be a problem, as, during the development of this work, FLUIDOS's auto-discovery mechanism, which is running inside the drone, was not yet able to locate other clusters (such as the edge), thus making the establishing of new peerings not possible in such conditions. One way to resolve this issue is to create an additional component that tries to locate clusters that are not directly reachable by the drone, this is what has been done in this project with the "Discovery Agent".

6.5.1 Explaining the Discovery Agent logic

In order to locate the remote cluster (the network's edge), the Gateway present inside it has to be contacted, specifically the port on which FLUIDOS's REAR protocol is listening; this can be useful because, if the remote cluster exists, an HTTP 200 code will be returned to the Discovery Agent, otherwise the latter will periodically try to send other requests to this address until the desired response is received. After this procedure, the Discovery Agent can create a `KnownCluster` CR for the remote cluster, just as expected from FLUIDOS's usual behavior, furthermore, as soon as said CR is created, the program will keep contacting the same address in order to check if the remote cluster is still reachable; if the response is negative, the Discovery Agent will try again for a specific amount of times before deleting the `KnownCluster` CR and restarting the whole procedure from the start, otherwise, if the response is positive, the total number of current attempts will be set to 0, as the remote cluster is reachable again. The Discovery Agent takes three configuration parameters:

1. **Gateway address**: this represents the pair of address and port number that have to be contacted.
2. **Attempts**: the number of attempts after which the `KnownCluster` CR is deleted.
3. **Polling interval**: the time interval between each HTTP request from the Discovery Agent.

6.5.2 Implementing the discovery process

The first operation that the discovery process executes is to get the configuration parameters, after that a `DiscoveryAgent` struct (Listing A.5) is created in order to store and manage the configuration parameters, so that they can be used during the control loop. The last operation is to start the discovery control loop by calling periodically the `DiscoverNewClusters` function, which manages the logic of a single HTTP request. The implementation of the `DiscoverNewClusters` function can be seen by looking at Listing 6.5, it implements the exact logic that was already explained in Subsection 6.5.1, so it will start by sending an HTTP request to the address passed as configuration parameter, specifically the request will be sent to the endpoint `/api/v2/flavors`; if the response is an HTTP 200 code, a `KnownCluster` CR will be created, otherwise the control loop will keep executing this function, thus sending other requests, until a positive response is received or the number of total attempts has reached its maximum, in which case the `KnownCluster` will be deleted.

```go
func (agent *DiscoveryAgent) DiscoverNewClusters() error {

    response, err := http.Get("http://" + agent.Config.GatewayAddr +
    ↪ "/api/v2/flavors")

    if err != nil {
        return err
    }

    if response.StatusCode == 200 {

        agent.CurrentAttempt = 0
        err = agent.createKnownCluster()

        if err != nil {
            return err
        }
    }

    agent.CurrentAttempt++

    if agent.CurrentAttempt == agent.Config.TotalAttempts {

        err = agent.deleteKnownCluster()

        if err != nil {
            return err
        }
    }

    return nil
}
```

Listing 6.5: Implementation of the `DiscoverNewClusters` function

6.6 Offloading the workload between multiple clusters

This section shows the implementation of the Arbiter, which is the most important component in the system, by providing a thorough explanation about its purpose and logic.

6.6.1 Arbiter logic

There are multiple configuration parameters that the Arbiter needs:

1. **Drone IMSI**: the IMSI of the drone on which the Arbiter is executing.
2. **Prometheus URL**: the URL of the Prometheus endpoint located in the remote cluster.
3. **Delta**: the same delta value that was already discussed in Section 6.1.
4. **Scores scraping window**: this value will be used when executing the PromQL query to Prometheus; it represents the number of seconds to be considered when calculating the average score, more details about this will be provided later.
5. **Decision timeout**: the number of seconds between each cycle of the Arbiter control loop.
6. **Threshold**: the same threshold value that was already discussed in Section 6.1.
7. **Solver filters**: this is a file that contains the filters required by the `Solver` CR during the act of its creation.

The Arbiter logic is represented in Figure 6.8, as it can be seen it starts its execution by getting the required configuration parameters and by checking whether a `Solver` CR already exists, in that case the Arbiter proceeds to wait for new peerings to be established, otherwise it needs to create it by waiting until new `KnownCluster` CRs appears[6], at this point a new `Solver` is going to be created and the execution flow can proceed as normal; when new peerings are established, the Arbiter control loop can start. A single cycle of this loop can be presented as follows: the first operation is to execute the proper PromQL query to Prometheus in

[6]The same ones created by the Discovery Agent or by FLUIDOS's default auto-discovery.

order to get the average score that the Arbiter needs, then the offloading operation has to be performed according to the logic that was already explained in Section 6.1; after this is done, the cycle terminates its execution and another one will start soon after.

6.6.2 Outer control loop

The Arbiter control loop periodically calls the `MonitorScore` function, which is the one that actually implements the logic of a single iteration; all these iterations are separated by a certain amount of time that is specified through the configuration parameter known as "Decision timeout", this is done in order to wait for the channel score to actually change over time, otherwise the Arbiter would keep sending useless queries to Prometheus that would only add unnecessary workload inside the system, as it is unlikely that said score changed much right after the last query. The first operation performed by the `MonitorScore` is to invoke the `executeQuery` function, which executes the PromQL query in order to get the average score.

6.6.3 Executing the PromQL query to Prometheus

The query that the `executeQuery` function sends will compute the average score according to the logic that was already discussed in Section 6.1, specifically this query has to be formulated by using the IMSI value of the drone on which the Arbiter is executing as the label, this is needed in order to get the score values associated to the drone. The configuration parameter called "Scores scraping window" is going to be used here as the time interval to consider when computing the average score, for example, if this parameter is set to 10, then the average will be computed by considering all the score values generated in the last 10 seconds. If the response to the query is positive the average score will be returned by the function to its caller, which is `MonitorScore`, otherwise, if the query failed due to the fact that Prometheus is unreachable[7], a local offloading operation will be forced, thus moving the entire workload to the drone.

6.6.4 Performing the offloading operation

The average score returned by the `executeQuery` function will now be used in order to understand what kind of offloading operation is needed (Figure 6.8). Listing 6.6 shows the implementation of the offloading functions, they were both implemented according to the logic of the solution presented in Subsection 6.3.1.

[7]This can be caused, for example, by the fact that the drone lost its connection to the 5G cell.

```go
// This function performs the remote offloading operation (from Drone
    ↪ to Edge).
func ExecuteRemoteOffloading(ctx context.Context, client client.
    ↪ Client) error {

    err := updateTaint(ctx, client, Remove)

    if err != nil {
        return err
    }

    return changeOffloadingStrategy(ctx, client, offloadingv1alpha1.
    ↪ RemotePodOffloadingStrategyType)
}

// This function performs the local offloading operation (from Edge
    ↪ to Drone).
func ExecuteLocalOffloading(ctx context.Context, client client.Client
    ↪ ) error {

    err := changeOffloadingStrategy(ctx, client, offloadingv1alpha1.
    ↪ LocalPodOffloadingStrategyType)

    if err != nil {
        return err
    }

    return updateTaint(ctx, client, Add)
}
```

Listing 6.6: Implementation of the offloading functions

After the execution of the appropriate action, the current iteration of the Arbiter control loop terminates; the next one will start after the amount of seconds specified by the "Decision timeout" configuration parameter.

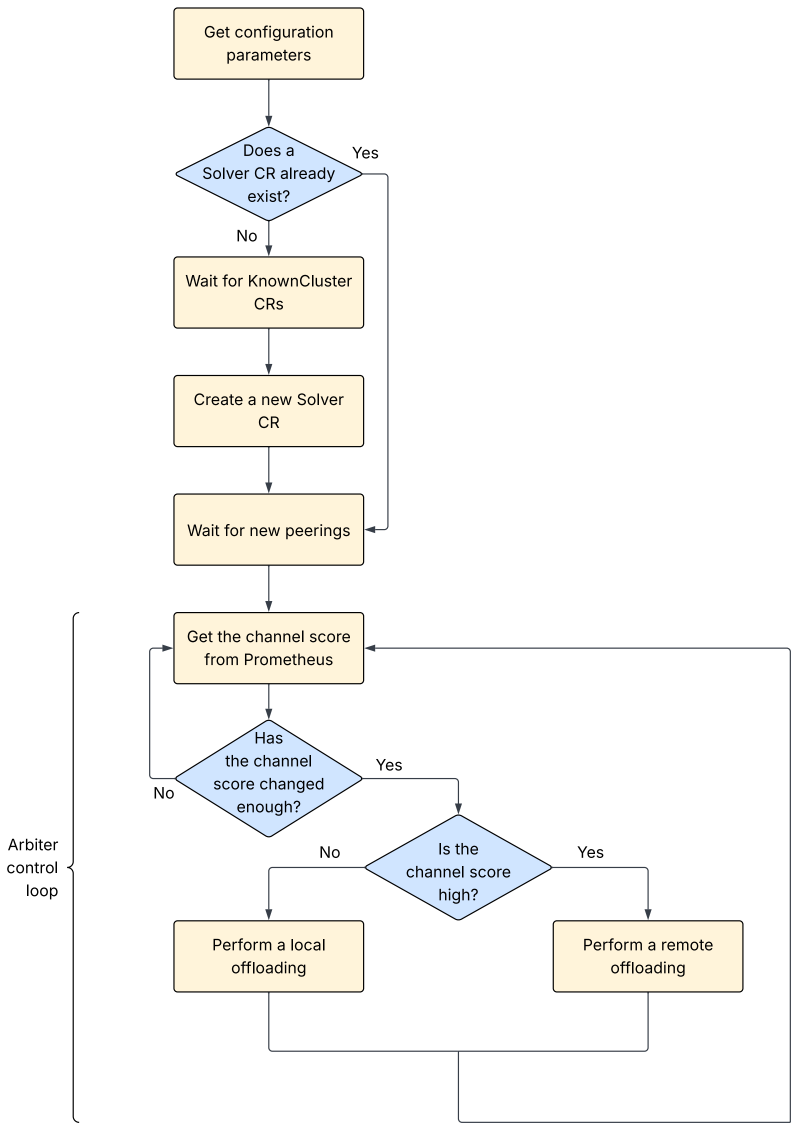

Get configuration parameters
Does a Solver CR already exist?
Yes
No
Wait for KnownCluster CRs
Create a new Solver CR
Wait for new peerings
Get the channel score from Prometheus
Has the channel score changed enough?
No
Yes
Is the channel score high?
No
Yes
Perform a local offloading
Perform a remote offloading
Arbiter control loop

Figure 6.8: Flowchart representing the Arbiter logic

Chapter 7

Experimental evaluations

This chapter will discuss the results obtained by testing the system's capabilities in a real-world scenario. It will be first presented the testing environment used for the development of the video that shows a functioning demo of the system, then the time required by both of the offloading operations will be discussed. In conclusion, the chapter will make a comparison between the performances obtained by using either the channel score or the 5G signal strength that was already presented in Subsection 6.4.4 for what concerns the Arbiter reaction time to changes in the channel quality.

7.1 Setup of the test environment

All the tests were conducted in the "Consiglio Nazionale delle Ricerche" (CNR) facility located at Politecnico di Torino by using a 5G cell that was deployed there. The device used to simulate the drone was a laptop running Ubuntu 20.04 LTS on which the kernel has been patched in order to allow a connection to the 5G cell through another device used as a UE, which was a Quectel modem (RG50xQ/RM5xxQ Series) attached to the laptop via USB. The network's edge was simulated by a mini-PC running the same version of Ubuntu used for the laptop, which was connected to the 5G cell through a router. By looking at Figure 7.1, the overall testing setup can be seen. The Arbiter was configured with the following configuration parameters:

- `Delta`: 1
- `Scores scraping window`: 5 seconds
- `Decision timeout`: 10 seconds
- `Threshold`: 1.5

In order to simulate the workload, a simple `nginx` job running inside an offloaded Liqo namespace has been used. The local cluster is located on the laptop, the remote one is on the mini-PC; whenever an offloading operation occurs, the `nginx` job will be offloaded from one cluster to the other as needed.

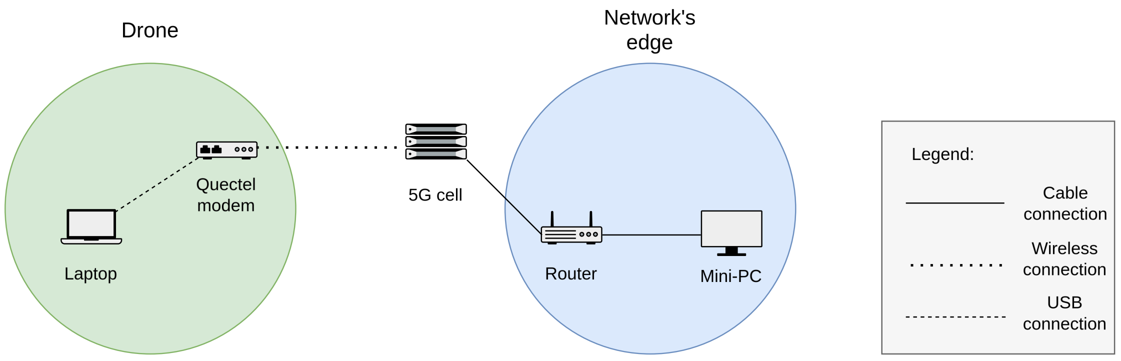


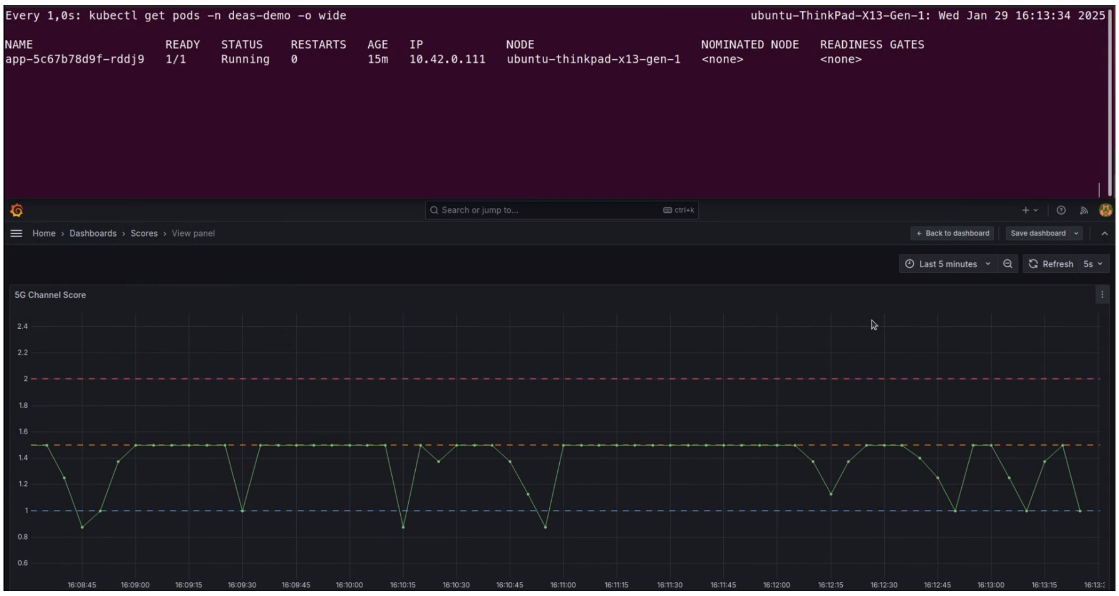
Figure 7.1: Testing environment

7.2 Simulating a workload offloading scenario

The tests were conducted by physically moving the laptop used to simulate the drone; whenever said laptop was moved away from the 5G cell, it simulated a decrease of the channel quality, on the contrary, if it was moved closer to the 5G cell, an increase of the channel quality would have occurred. The execution flow described in the following Subsections represents the situation from the laptop's standpoint.

7.2.1 Performing a remote offloading

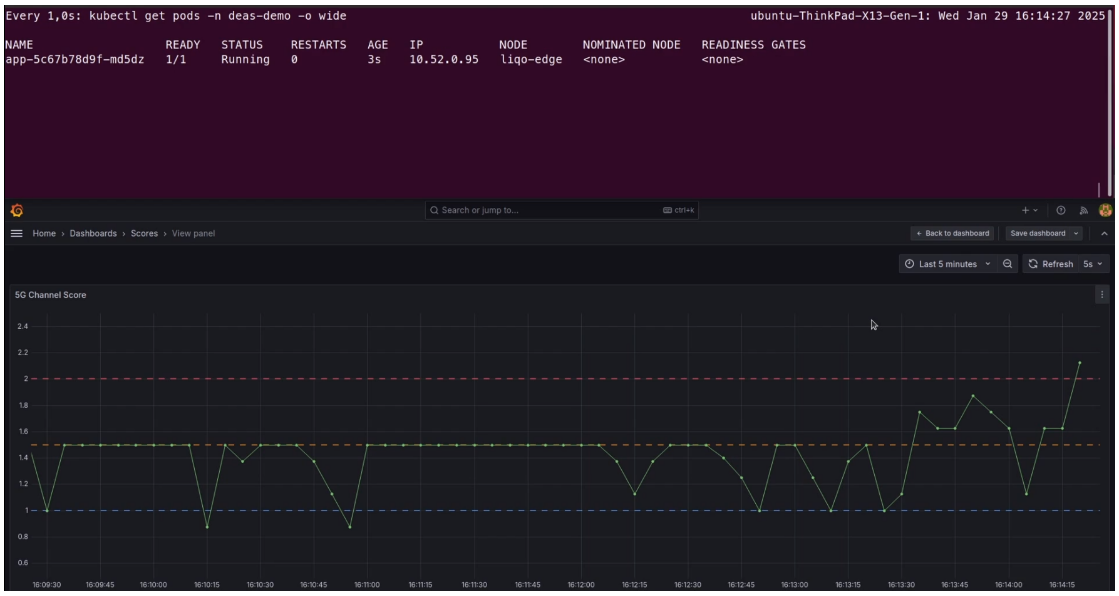
The simulation started by executing the aforementioned `nginx` job on the local cluster, as the channel quality never surpassed the value of 2 (the red line), this can be seen by looking at Figure 7.2; specifically, the task was executing on a node called `ubuntu-thinkpad-x13-gen-1`. The first thing to do was to simulate a remote offloading by placing the `nginx` job on the remote cluster, which, as already mentioned, was simulated by a mini-PC. As soon as the laptop was moved closer to the 5G cell, we experienced an increase of the channel quality, just as expected. In Figure 7.3, it is possible to observe what has been described before: the channel quality surpassed 2, this triggered a remote offloading that moved the task to the remote cluster, in fact it can be seen that now the node on which this job is running is called `liqo_edge`, this is the virtual node that Liqo created to represent the remote cluster.



```
Every 1,0s: kubectl get pods -n deas-demo -o wide                                           ubuntu-ThinkPad-X13-Gen-1: Wed Jan 29 16:13:34 2025

NAME                    READY   STATUS    RESTARTS   AGE   IP            NODE                        NOMINATED NODE   READINESS GATES
app-5c67b78d9f-rddj9    1/1     Running   0          15m   10.42.0.111   ubuntu-thinkpad-x13-gen-1   <none>           <none>
```

Figure 7.2: Task executing on the local cluster



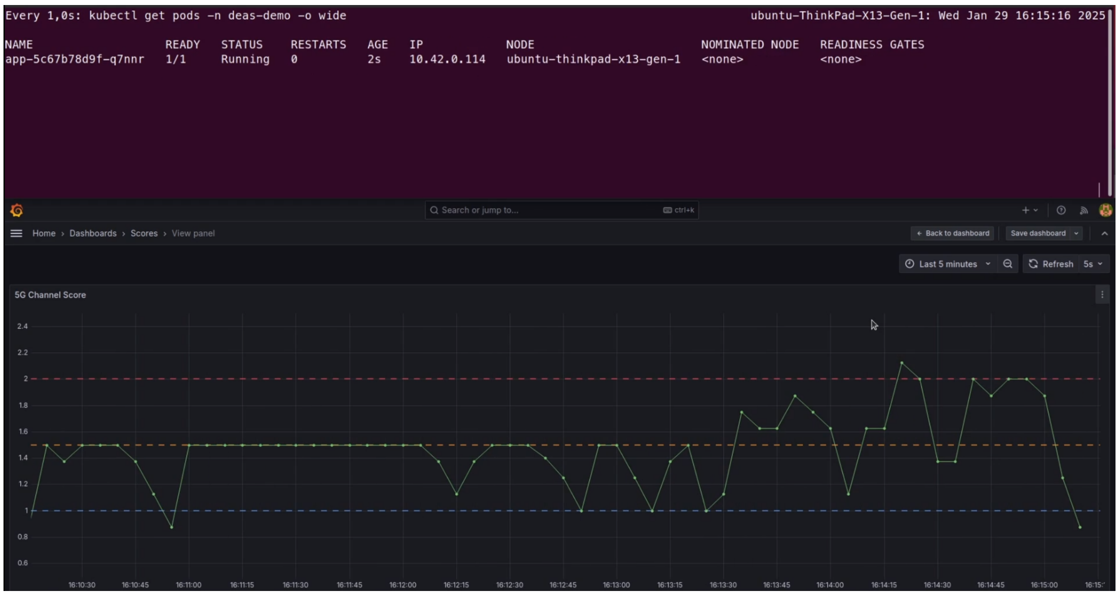
```
Every 1,0s: kubectl get pods -n deas-demo -o wide                                           ubuntu-ThinkPad-X13-Gen-1: Wed Jan 29 16:14:27 2025

NAME                    READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINATED NODE   READINESS GATES
app-5c67b78d9f-md5dz    1/1     Running   0          3s    10.52.0.95   liqo-edge   <none>           <none>
```

Figure 7.3: Task executing on the remote cluster

7.2.2 Performing a local offloading

The last thing to do was to simulate a local offloading by moving the laptop away from the 5G cell, this situation is represented in Figure 7.4. As it can be seen, the channel quality is now below 1 (the blue line), this has triggered a local offloading that moved the task back to the local cluster, in fact the node on which this job is executing is again `ubuntu-thinkpad-x13-gen-1`. By looking at the obtained results, we have assured that the system can properly work in real environments, as both of the offloading operations are performed efficiently just as intended.



```
Every 1,0s: kubectl get pods -n deas-demo -o wide                                  ubuntu-ThinkPad-X13-Gen-1: Wed Jan 29 16:15:16 2025

NAME                    READY   STATUS    RESTARTS   AGE   IP            NODE                        NOMINATED NODE   READINESS GATES
app-5c67b78d9f-q7nnr    1/1     Running   0          2s    10.42.0.114   ubuntu-thinkpad-x13-gen-1   <none>           <none>
```

Figure 7.4: Task executing again on the local cluster

7.3 Comparing the offloading operations: local and remote

By looking at Figure 7.5, the obtained performances for both of the offloading operations can be seen. The time interval that was considered is the one that starts as soon as their respective functions are called (Listing 6.6) and ends when the workload starts to run on the other cluster; in this graphical representation, 100 different time values have been considered for each one of them. The obtained results show that the execution time for both of these operations is around 1 second, with the local offloading operation being slightly slower than the remote one, this could be caused by Liqo and/or Kubernetes internal functioning, as both of these operations perform the exact same actions, just in reverse order. In conclusion, we could say that the system can maintain good performances for what concerns the workload offloading.

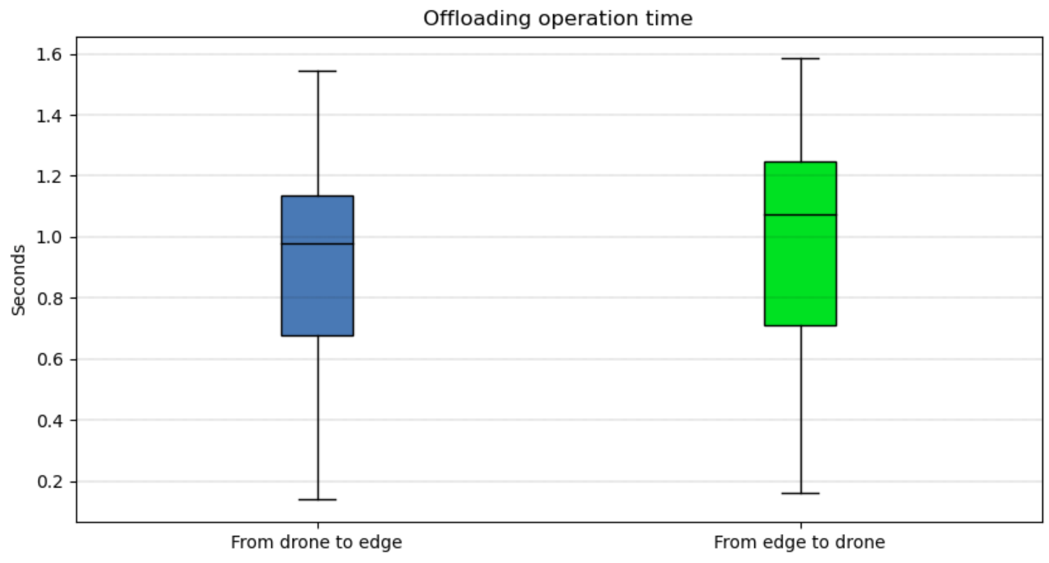


Figure 7.5: Time required by both of the offloading operations

7.4 Comparing the Arbiter reaction time: 5G signal strength and channel score

In Subsection 6.4.4, we already discussed about the possible usage of the 5G signal strength in order to replace the channel score used by the system; the main advantage of this approach is the increase in responsiveness, as this "local score" would be fetched by the drone locally, without the need to go through the network multiple times, as a consequence the Arbiter should be able to react even quicker to any change in the quality of the channel[1]. The detailed approach would be the following: the component called "dB exporter" (Figure 6.7) should periodically fetch the 5G signal strength, represented in decibels, from the Quectel modem that is connected to the local device through the appropriate AT commands, this value should be made available to Prometheus as a metric, at this point the Arbiter can use the metric for the offloading process, just as always. Considering that, as already mentioned, this was not part of the implementation process, in fact its behavior has only been simulated by creating a component that periodically opens a file containing a random series of possible values in decibels that could have been returned by the modem. By looking at Figure 7.6, it can be seen a

[1]A change in the channel quality could be the transition from a positive score to a negative one or vice-versa.

graphical comparison of the performances obtained by the two alternatives for what concerns the Arbiter reaction time; just as always, 100 different time values have been considered for both of them, specifically the time interval that was considered is the one that starts as soon as the channel quality changes and ends when the workload starts to run on the other cluster due to an offloading operation triggered by the aforementioned change in the channel quality. As can be seen, by using the 5G signal strength, we could achieve great performances compared to the channel score approach, as, in the first case, the Arbiter can react to changes in the channel quality in around 8 seconds, while the second approach makes the Arbiter react in around 37 seconds, this is a great improvement. It is worth mentioning though that this graph compares real data, obtained by using the channel score, with data obtained through a simulation, which is the case of the 5G signal strength, this means that the results that were previously discussed are just speculative and should not be considered as an actual fact. The 5G signal strength is not always the best option though, as this value is just an indication about the channel quality and not a precise measurement; on the contrary the channel score provides a more accurate estimation, as it is derived by multiple real 5G metrics instead of just a single indicator.

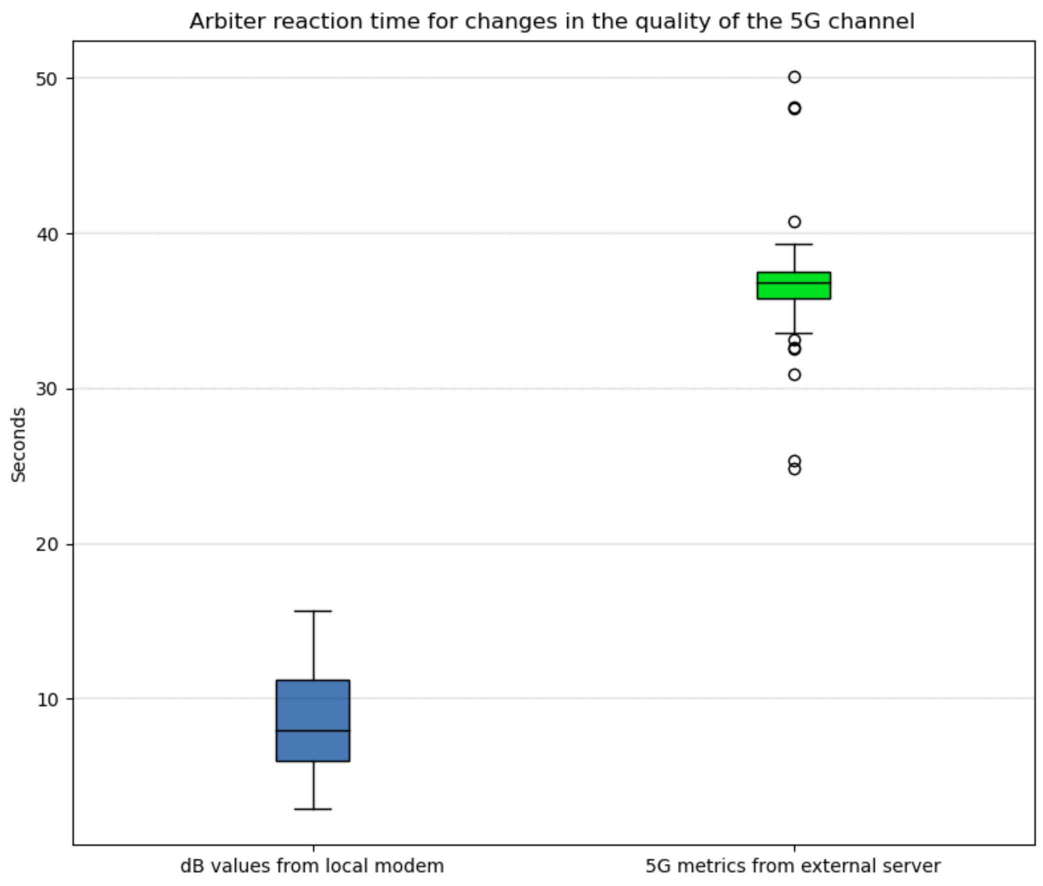


Figure 7.6: Arbiter reaction time

Chapter 8

Conclusions

The purpose of this work was to enhance FLUIDOS capabilities by providing the appropriate support for ensuring seamless operations even in fluctuating network conditions through the DEAS open call. The goal was to develop an infrastructure capable of ensuring dynamic offloading capabilities in scenarios where the workload is periodically moved between IoT devices, such as drones, and the network's edge in 5G networks, depending on the network's quality; the main advantage is that the drones' workload can be reduced by offloading some of their tasks to the network's edge, and, by doing so, all the unused computational power present at the edge will not be wasted, this allows to optimize the resources and the overall energy consumption of the system by creating a responsive computing continuum. The offloading process needed to perform the following logic: if the channel quality is good enough, the workload has to be offloaded from the drones to the network's edge, on the contrary, if the quality is bad, the workload has to return to the drones. Two main strategies were taken into account in order to implement the offloading process; the first one just involved the usage of Kubernetes and Liqo native features, while the second considered the usage of the Kubernetes Descheduler. Even though the Descheduler is particular convenient when it comes to offload tasks that are already executing (there is no need to manually evict Pods), the first one is preferable, as no additional components need to be installed, furthermore the offloading logic can be kept simple. The functioning of the system is the following: the remote cluster (the edge) periodically collects an indicator of the channel quality from an external metrics server, which will be stored as a Prometheus metric; at this point, the local cluster (the single drone) will fetch this score from the remote cluster by contacting its Prometheus endpoint, after this is done the Arbiter, a component responsible of the offloading operation that runs inside the local cluster, will perform the offloading decision according to the logic that was previously discussed. The final system has been tested in the CNR facility located at Politecnico di Torino by using a 5G cell; the drone (the local cluster) has been simulated by a laptop, the network's edge (the remote cluster) by a mini-PC and the workload by a simple `nginx` job. The workload started its execution on the local cluster due to a bad channel quality, which has been simulated by physically moving away the laptop from the 5G cell, so the first operation that needed to be performed was a remote offloading. In order to simulate an increase of the channel quality, the laptop has been moved closer to the 5G cell, this triggered a remote offloading operation that placed the workload on the remote cluster, just as expected. The last operation that needed to be tested was the local offloading, in fact, by moving again the laptop away from the 5G cell, the channel quality gradually decreased, hence the workload has been placed for execution on the local cluster. Additionally, the performance level of the offloading operations has been evaluated, in fact both of them allow to dynamically move the workload from one cluster to the other in about 1 second. Another alternative approach has been considered, one that does not rely on 5G metrics fetched from an external metrics server, but one that instead allow drones to directly fetch the value in decibels of the 5G signal strength locally by contacting the local modem; even though this other approach has only been simulated with no real implementation, the performance level could be drastically increased compared to the external metrics server strategy (8 seconds for the first one against 37 seconds for the second). Considering the obtained results and the good performance level, we can achieve an autonomous, adaptive, and efficient workload distribution that allows to offload tasks between drones and the edge by using the quality of the 5G channel as a decision criterion, thus creating a responsive computing continuum; a demo showing the system's functioning is available online [31].

8.1 Future directions

A possible future direction for the project could be the actual implementation of an additional component that fetches the 5G signal strength by making it available on Prometheus alongside the channel score fetched from the external metrics server, so that these two different paradigms could alternatively be used by the Arbiter as needed. An example of this is where the user wants to give priority to a scenario where responsiveness is more important than the accuracy of the channel quality, in that case the 5G signal strength could be a better option, on the contrary, if the user prioritizes accuracy over responsiveness, the external metrics server would be the preferred alternative. Another possible future work is the dynamic configuration of the optimal values for the threshold and delta, as it could be difficult to understand if the inserted values are correct for the specific use case, this would make the system even more efficient by reducing to a minimum the errors and the need for manual configurations.

Appendix A

Listings

```
{
    "series": [
        {
            "values": [
                [
                    "2024-10-01T07:31:43.428469Z",
                    "0x000e",
                    "0xe000",
                    "001,01",
                    "Connected",
                    "gNB-Sma-RTy"
                ]
            ]
        }
    ]
}
```

Listing A.1: Example of JSON object returned by the endpoint `/amf_gnb_stats`

```
{
    "series": [
        {
            "values": [
                [
                    "2025-01-24T10:55:44.531977Z",
                    "5GMM-REGISTERED",
                    "0x000e",
                    1,
                    14680064,
                    "",
                    "001010000000025",
                    "001, 01",
                    "1"
                ]
            ]
        }
    ]
}
```

Listing A.2: Example of JSON object returned by the endpoint `/amf_ue_stats`

```
{
    time: "2024-10-01T11:33:37.002967Z"
    Score: 8.4
    Ran_ngap_ue_id: "6"
}
```

Listing A.3: Example of JSON object returned by the endpoint `/ue_mac_stats/{gnb_id}/imsi/{imsi}/score`

```go
package scorecollector

import "github.com/prometheus/client_golang/prometheus"

// ScoreCollector implements the Collector interface.
type ScoreCollector struct {
	ScoreValue   *prometheus.Desc
	apiServerUrl string
}

// This map associates IMSIs with their score values.
type ScoreMap map[string]float64

// These structs will be used to store the result returned by
// the first two endpoint, "amf_ue_stats" and "/amf_gnb_stats".
type Series struct {
	Values [][]interface{} `json:"values"`
}

type Data struct {
	Series []Series `json:"series"`
}

// This struct will be used to store the result returned by
// the endpoint "/ue_mac_stats/{gnb_id}/imsi/{imsi}/score".
type Score_struct struct {
	Time           string  `json:"time"`
	Score          float64 `json:"Score"`
	Ran_ngap_ue_id string  `json:"Ran_ngap_ue_id"`
}
```

Listing A.4: Data structures used to store the JSON objects returned by the endpoints

```go
package manageClusters

import (
    "context"

    "sigs.k8s.io/controller-runtime/pkg/client"
)

// This struct will be used to interact with k8s functions.
type K8sAgent struct {
    Client  client.Client
    Context context.Context
}

// This struct will be used to manage configuration parameters.
type DiscoveryConfig struct {
    GatewayAddr     string
    TotalAttempts   int
    PollingInterval int
}

// This struct will be used to manage the Discovery control loop life
   ↪ -cycle.
type DiscoveryAgent struct {
    K8s            *K8sAgent
    Config         *DiscoveryConfig
    CurrentAttempt int
}
```

Listing A.5: Data structures used to manage the discovery process

```go
package scoremonitor

import (
    "github.com/prometheus/client_golang/api"
    v1 "github.com/prometheus/client_golang/api/prometheus/v1"
    "sigs.k8s.io/controller-runtime/pkg/client"
)

// This struct will be used to interact with Prometheus when
  ↪ performing a PromQL query.
type PrometheusAgent struct {
    client api.Client
    api    v1.API
}

// This struct will be used to store configuration data.
type ConfigParams struct {
    Imsi              string
    PrometheusUrl     string
    Delta             float64
    ScoresWindow      string
    DecisionTimeout   int
    Threshold         float64
    SolverFiltersPath string
}

// This struct will be used to manage the Arbiter control loop life-
  ↪ cycle.
type ScoreMonitorAgent struct {
    prometheusClient *PrometheusAgent
    client           client.Client
    isFirstIteration bool
    isRemote         bool
    imsi             string
    prometheusUrl    string
    delta            float64
    scoresWindow     string
    threshold        float64
}
```

Listing A.6: Data structures used to manage the offloading process inside the Arbiter

Bibliography

[1] Auday Al-Dulaimy et al. «The computing continuum: From IoT to the cloud». In: *Internet of Things* 27 (2024) (cit. on pp. 2, 4).

[2] Kaiyuan, Chen, Y. Liang, N. Jha, J. Ichnowski, M. Danielczuk, J. Gonzalez, J. Kubiatowicz, and K. Goldberg. «Fogros: An adaptive framework for automating fog robotics deployment». In: *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)* (Aug. 2021), pp. 2035–2042 (cit. on p. 7).

[3] J. Ichnowski et al. «Fogros2: An adaptive platform for cloud and fog robotics using ros 2». In: *2023 IEEE International Conference on Robotics and Automation (ICRA)* (July 2023), pp. 5493–5500 (cit. on p. 7).

[4] K. Chen, R. Hoque, K. Dharmarajan, E. LLontopl, S. Adebola, J. Ichnowski, J. Kubiatowicz, and K. Goldberg. «Fogros2-sgc: A ros2 cloud robotics platform for secure global connectivity». In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Dec. 2023) (cit. on p. 7).

[5] A. Chebaane, S. Spornraft, and A. Khelil. «Container-based task offloading for time-critical fog computing». In: *2020 IEEE 3rd 5G World Forum (5GWF)* (Sept. 2020), pp. 205–211 (cit. on p. 7).

[6] R. Anand, J. Ichnowski, C. Wu, J. M. Hellerstein, J. E. Gonzalez, and K. Goldberg. «Serverless multi-query motion planning for fog robotics». In: *2021 IEEE International Conference on Robotics and Automation (ICRA)* (2023), pp. 7457–7463 (cit. on p. 7).

[7] T. V. Doan, Z. Fan, G. T. Nguyen, H. Salah, D. You, and F. H. P. Fitzek. «Follow me, if you can: A framework for seamless migration in mobile edge cloud». In: *IEEE Conference on Computer Communications (INFOCOM)* (July 2020), pp. 1178–1183 (cit. on p. 8).

[8] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis. «Live service migration in mobile edge clouds». In: *IEEE Wireless Communications* 25 (Feb. 2018), pp. 140–147 (cit. on p. 8).

[9] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys)* (2015) (cit. on p. 9).

[10] Ferenc Hámori. *The History of Kubernetes on a Timeline*. June 2018. URL: `https://blog.risingstack.com/the-history-of-kubernetes/` (cit. on p. 9).

[11] Melissa Sussmann. *Five reasons why every CIO should consider Kubernetes*. Dec. 2023. URL: `https://www.sumologic.com/blog/why-use-kubernetes/` (cit. on p. 9).

[12] *Kubernetes architecture*. URL: `https://kubernetes.io/docs/concepts/architecture/` (cit. on p. 10).

[13] *Objects in Kubernetes*. URL: `https://kubernetes.io/docs/concepts/overview/working-with-objects/` (cit. on p. 13).

[14] *Kubernetes deployments*. URL: `https://kubernetes.io/docs/concepts/workloads/controllers/deployment/` (cit. on p. 15).

[15] *Kubernetes services*. URL: `https://kubernetes.io/docs/concepts/services-networking/service/` (cit. on p. 16).

[16] *Kubernetes Descheduler*. URL: `https://github.com/kubernetes-sigs/descheduler.git` (cit. on p. 17).

[17] *Prometheus*. URL: `https://github.com/prometheus/prometheus` (cit. on p. 18).

[18] *Prometheus overview*. URL: `https://prometheus.io/docs/introduction/overview/` (cit. on p. 18).

[19] *Grafana*. URL: `https://github.com/grafana/grafana` (cit. on p. 20).

[20] *Prometheus Operator*. URL: `https://github.com/prometheus-operator/prometheus-operator` (cit. on p. 20).

[21] *Example of ServiceMonitor*. URL: `https://github.com/prometheus-operator/kube-prometheus/blob/main/manifests/prometheus-serviceMonitor.yaml` (cit. on p. 21).

[22] *Liqo*. URL: `https://github.com/liqotech/liqo` (cit. on p. 22).

[23] *Liqo peering*. URL: `https://docs.liqo.io/en/v0.10.3/features/peering.html` (cit. on p. 23).

[24] *Namespace offloading in Liqo*. URL: `https://docs.liqo.io/en/v0.10.3/features/offloading.html` (cit. on p. 25).

[25] *FLUIDOS*. URL: `https://fluidos.eu/` (cit. on p. 26).

[26] *FLUIDOS REAR protocol.* URL: https://github.com/fluidos-project/REAR (cit. on p. 27).

[27] *Neuropil.* URL: https://www.neuropil.org/ (cit. on p. 28).

[28] *1st open call winners.* URL: https://fluidos.eu/1st-open-call-winners/ (cit. on p. 28).

[29] *Go for Cloud and Network Services.* URL: https://go.dev/solutions/cloud (cit. on p. 34).

[30] *General rules for Prometheus exporters.* URL: https://prometheus.io/docs/instrumenting/writing_exporters/ (cit. on p. 40).

[31] *DEMO - DEAS project: Enhancing Dynamic Workload Offloading with FLUIDOS and 5G.* URL: https://www.youtube.com/watch?v=H9xtZDQkKT0 (cit. on p. 58).